symmetry

MDPI

Article

Understanding Review Expertise of Developers: A Reviewer Recommendation Approach Based on Latent Dirichlet Allocation

Jungil Kim [1] and Eunjoo Lee [2,*]

[1] Department of Software Technology Laboratory, Kyungpook National University, Daegu 41566, Korea; jikim424@gmail.com
[2] School of Computer Science and Engineering, Kyungpook National University, Daegu 41566, Korea
* Correspondence: ejlee@knu.ac.kr; Tel.: +82-53-950-7548

Received: 21 March 2018; Accepted: 16 April 2018; Published: 17 April 2018

Abstract: The code reviewer assignment problem affects the reviewing time of a source code change. To effectively perform the code review process of a software project, the code reviewer assignment problem must be dealt with. Reviewer recommendation can reduce the time required for finding appropriate reviewers for a given source code change. In this paper, we propose a reviewer recommendation approach based on latent Dirichlet allocation (LDA). The proposed reviewer recommendation approach consists of a review expertise generation phase and a reviewer recommendation phase. The review expertise generation phase generates the review expertise of developers for topics of source code changes from the review history of a software project. The reviewer recommendation phase computes the review scores of the developers according to the topic distribution of a given source code change and the review expertise of the developers. In an empirical evaluation of five open source projects, we confirm that the proposed reviewer recommendation approach obtains better average top-10 accuracy than existing reviewer recommendation approaches.

Keywords: software engineering; machine learning; reviewer recommendation

1. Introduction

Software project developers often perform code reviews to reduce maintenance costs and ensure sustainability of a software project [1,2]. Code review is a manual inspection of the source code that is performed by developers. Inefficient logic and latent bugs in the source code significantly increase the maintenance costs of a software project. Although code reviews are time consuming, it is beneficial to detect the defects in the source code at an early software development stage [1,3–9].

Recently, a modern code review process has been adopted to more efficiently perform code reviews in both open source projects and industry software projects [10]. The modern code review process is a tool-based, lightweight code review mechanism [11]. When a source code change is submitted, the code review process for the submitted source code change begins. The author of the submitted source code change and the project manager request reviews from developers who are suitable to review the source code change. The developers who accept the review request perform the review of the source code change and provide valuable feedback to the author. The author improves the source code change by referring to the feedback provided by the reviewers.

The reviewer assignment is a major challenge in performing the code review process rapidly and successfully [2,5–7,10,12,13]. Thongtanunam et al. investigated the impact of the reviewer assignment problem by conducting an exploratory study in open source projects. They found that reviews with the code reviewer assignment problem took much more time to complete the review process. In particular,

as open source project development is usually developed by the voluntary participation of many developers, it is difficult to rapidly find appropriate reviewers who have the approximate expertise for a given source code change. Hence, the author of a source code change and the project manager devote much effort to finding appropriate reviewers.

A code reviewer recommendation tool can be helpful in reducing the reviewer assignment time. In previous research in the software engineering field, reviewer recommendation approaches have been considered for resolving the reviewer assignment problem. Thongtanunam et al. proposed a file location-based reviewer recommendation approach called REVFINDER. REVFINDER recommends reviewers based on the similarity of file paths. Developers who often reviewed the files located in paths similar to the path of the changed files in a given review request are more likely recommended as suitable reviewers. Zanjani et al. proposed a reviewer recommendation approach, cHRev, which considers the review count of developers to source files as a basic factor for computing review expertise. cHRev recommends as suitable reviewers the developers who reviewed the changed files in the recent past. Both REVFINDER and cHRev consider only file-level features, such as the name and location of files, as a factor for computing the review expertise of developers. The approaches do not consider the information of source code changes that are actually reviewed by developers to determine the review expertise of developers.

In this paper, we present a way to compute the review expertise of developers at the source code change level. More specifically, we compute the review expertise of developers based on the source code changes reviewed by the developer. The intuition of this idea assumes that developers typically tend to review the source code changes that are associated with their expertise or that interest them. Therefore, to more widely understand review expertise, it is necessary to derive the review expertise of developers from source code changes that are frequently reviewed by the developer. Similar to a typical text document, a source code change submitted to a software project consists of textual content that belongs to one or more specific topics that are related to some functionality of a software project [14]. Latent Dirichlet allocation (LDA) is a generative statistical model [15]. Previous studies on text analysis [16–18] used LDA to extract topic distribution from textual data.

Based on LDA, we propose a novel reviewer recommendation approach. The proposed reviewer recommendation approach consists of a review expertise generation phase and a reviewer recommendation phase. The review expertise generation phase extracts the K topics of source code changes reviewed by developers using LDA and computes the review expertise of the developers for K topics. The reviewer recommendation phase computes the review scores of the developers with their review expertise for K topics and recommends the top N reviewers according to the computed review scores. In an empirical study of five open source projects, we compared the proposed reviewer recommendation approach with REVFINDER and cHRev. For the top-10 recommendations by the reviewer recommendation approaches, the proposed reviewer recommendation approach obtained better recommendation results than REVFINDER and cHRev. The proposed reviewer recommendation approach, REVFINDER, and cHRev achieved 64%, 60%, and 54% of the average top-10 accuracy, respectively.

The remainder of this paper is organized as follows. First, we introduce the basic background of the code review process and LDA in Section 2. In Section 3, we present a motivation example of this study. We propose the process of our reviewer recommendation approach in Section 4. In Section 5, we report the results of the empirical evaluation performed for evaluating the effectiveness of the proposed reviewer recommendation approach. In Section 6, we introduce related work. Finally, we conclude this study with future research directions in Section 7.

2. Background

2.1. Code Review Process

Recent software projects actively employ the code review process [19]. The code review process begins with the reviewer assignment. When a new review request for a source code change is submitted

to a software project, a reviewer assigner ought to select appropriate reviewers to review the submitted source code change. The selected reviewers read the source code change and then send the review outcomes to the author in order to improve the quality of the source code change. The author modifies the source code change again according to the received review outcomes and then requests further code reviews for the newly modified source code change. Such a code review process is repeated until the quality of the submitted source code change is satisfactory.

The role of the reviewer assigner is important in order to efficiently perform the code review process [7]. If a reviewer assigner assigns an unsuitable reviewer who does not deeply understand the given source code change, the author of the source code change may obtain poor review outcomes. The poor review outcomes cause unintended modifications and make the review process repetitious. This may hinder the improvement of the source code change and delay the code review process. Therefore, a reviewer assigner must closely understand the review expertise of the developers to avoid failure of the reviewer assignment.

2.2. Topic Modeling

Topic modeling is a statistical model based on an unsupervised machine-learning algorithm [20]. Various topic modeling algorithms, including latent semantic indexing (LSI) [21], probabilistic LSI (pLSI) [22], and LDA [15], were proposed to extract topics from text documents. LDA is a generative statistical model and the most recently proposed topic modeling algorithm among them.

LDA assumes that a text document contains a mixture of several topics; various words in a text document are related to their topic. For example, words such as "computer", "network", and "software" typically appear in text documents related to computer science. LDA extracts topics based on the co-appearances of words in a collection of text documents. In LDA, a topic is defined as a set of words that are semantically related to each other. Each word in a text document is assigned to one of several topics. Each text document has a word-topic vector, which indicates the topic assignments of its words. The topic distribution of each text document is computed according to its word-topic vector.

Previous studies on software engineering often used LDA for analysis of unstructured data similar to a text document. Xie et al. [17] used LDA to extract topics from bug reports. Chen et al. [18] used LDA to extract topics from source files. In this paper, we used LDA to extract topics from source code changes.

3. Motivation

In this study, we assumed that developers will typically review source code changes with topics in which they are interested. To confirm the validity of this assumption, we empirically analyzed the review history of an open source project developed on GitHub (https://github.com). GitHub is a web-based software development hosting service that provides a pull-based development environment. In GitHub, a developer submits a pull request to commit his/her own source code changes to a software project. A pull request contains source code changes and its detailed description. A submitted pull request is committed to the source code repository of a software project after it is reviewed and approved by several participants in the project. A large number of open source projects are developed on GitHub. Thus, we were able to easily obtain the review history of source code changes for an open source project on GitHub.

We performed an empirical analysis of the review history of the bitcoin project (https://github.com/bitcoin). The bitcoin project is an open source project that maintains and releases a bitcoin client software. Many developers have participated in the bitcoin project. First, we collected the review history of the source code changes of the bitcoin project from the GHTorrent project [23]. The GHTorrent project provides various development history data of the open source projects collected from GitHub for research purposes. Second, we extracted topic distributions of the collected source code changes using an LDA implementation of MALLET (http://mallet.cs.umass.edu/). LDA has three parameters: K, alpha, and beta. K is the number of topics to be extracted, alpha is the mixture

of topics per document, and beta is the mixture of words per topic. The parameter K is sensitive in analyzing the topics of a software project because a value of K that is too large or too small will hinder exact topic generation [24]. Hence, according to [24], we set the value of K to 20. The values of alpha and beta were set to 5.0 and 0.01, respectively. The details of the process of topic extraction are described in the following section. Finally, after extracting the topics of all of the collected source code changes, we manually investigated whether developers often reviewed source code changes that had similar topic distributions.

Table 1 shows three reviewed commits found in the bitcoin project. "Review #" corresponds to a review identification number for a commit we arbitrarily assigned. "Commit sha" is a hash value of a commit with a length of 40. "Commit date" refers to the date on which the source code changes were committed. "Changed files" refer to the changed (reviewed) files in a commit. "Reviewer" corresponds to the developers who reviewed the changed files. Table 2 shows the topic distributions of the source code changes for 20 topics. Although the changed files are different, the topic distributions of their source code changes are quite similar to each other. The source code changes all have the same dominant topic. Topic 16 that is marked with bold in Table 1 constitutes over 97% of the topic distribution of the source code changes for all the three files.

Table 1. Details of reviewed commits in the *bitcoin* project.

Review #	**1**	**2**	**3**
Commit sha	08836972c093eb137e1c11eb 9596e7d12d600332	04da9306c62c062536a30 9e99178c9b742a01560	16d35eb228232ed53f87 cee233d0c8c3a9ca39eb
Commit date	2016-03-29	2016-04-13	2016-05-13
Changed files	src/rpcmisc.cpp	src/addressindex.h src/txdb.cpp	src/main.cpp src/main.h
Reviewers	UdjinM6 schinzelh	UdjinM6 schinzelh	UdjinM6 schinzelh

Table 2. The topic distributions of the reviewed source code changes in Table 1 for 20 topics.

Review #	**1**	**2**	**3**
Topic-0	0.000	0.000	0.000
Topic-1	0.002	0.001	0.000
Topic-2	0.001	0.001	0.000
Topic-3	0.001	0.001	0.000
Topic-4	0.002	0.001	0.000
Topic-5	0.002	0.001	0.000
Topic-6	0.001	0.001	0.000
Topic-7	0.002	0.001	0.000
Topic-8	0.001	0.001	0.000
Topic-9	0.001	0.000	0.000
Topic-10	0.003	0.001	0.000
Topic-11	0.003	0.001	0.000
Topic-12	0.002	0.001	0.000
Topic-13	0.001	0.000	0.000
Topic-14	0.002	0.001	0.000
Topic-15	0.002	0.001	0.000
Topic-16	0.970	0.987	0.999
Topic-17	0.001	0.001	0.000
Topic-18	0.001	0.000	0.000
Topic-19	0.001	0.001	0.000

4. LDA-Based Reviewer Recommendation Approach

4.1. Overall Process

Figure 1 shows the overall process of the reviewer recommendation approach that we propose in this paper. The proposed approach consists of two phases: the review expertise generation phase and the reviewer recommendation phase. In the review expertise generation phase, the review expertise of the developers is determined from the review history of source code changes. In the reviewer recommendation phase, reviewer candidates are recommended according to their review scores computed based on the result of the review expertise generation phase.

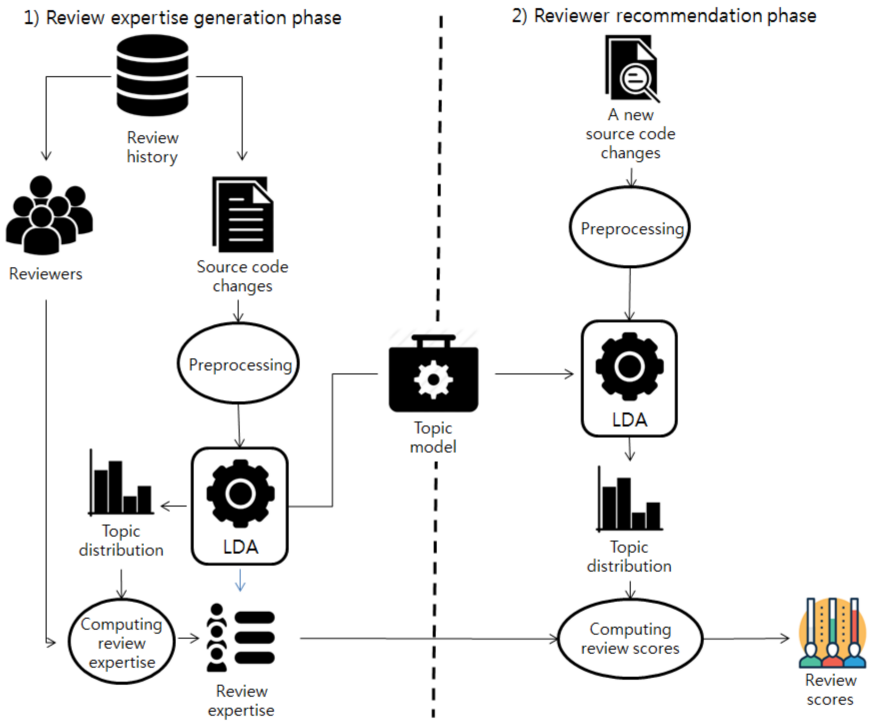

Figure 1. Overall process of our reviewer recommendation approach.

The review expertise generation phase is performed according to the following sub-steps. First, source code changes and a list of developers who reviewed the source code changes are extracted from the past review history of a software project. Second, the source code changes are preprocessed using several natural language processing techniques. After the preprocessing, the preprocessed source code changes are passed to LDA as input. LDA extracts the topic distributions of the given source code changes and generates a topic model. Lastly, the review expertise of the developers is computed.

The reviewer recommendation phase is performed according to the following sub-steps. When a new source code change is submitted, the submitted source code change is preprocessed. The topic distribution of the preprocessed new source code change is then inferred using the topic model generated at the review expertise generation phase. The review scores of the reviewer candidates are computed with the inferred topic distribution of the new source code change and the review expertise of the reviewer candidates. According to the computed review scores, the top-k reviewers are recommended. We describe in detail the processes of the proposed approach in the following subsections.

4.2. Review Expertise Generation Phase

The review expertise generation phase consists of three steps: preprocessing source code changes, extracting topics from source code changes, and computing the review expertise of developers. The following subsections describe each step in detail.

4.2.1. Preprocessing of Source Code Changes

A source code change is human-readable text data that are treated in a way similar to a text document. Figure 2 shows a source code change related to "http header cookie encoding". In general, frequently appeared words are less important in a text document. For example, words that are prepositions, articles, or conjunctions in English frequently appear in several texts and do not have a specific meaning. Hence, such words are typically treated as stop words in modeling an LDA model [25]. As the stop words, keywords, and operators frequently appear in several source code changes and are not related to any specific functionality of a software, they may affect the results of topic extraction by LDA. Therefore, a source code change needs to be preprocessed before being passed to LDA.

```
@@ -163,17 +197,24 @@ public String encode(Cookie cookie) {
* @return the corresponding bunch of Set-Cookie headers
*/
public List<String> encode(Iterable<? extends Cookie> cookies) {
- if (!checkNotNull(cookies, "cookies").iterator().hasNext()) {
+ Iterator<? extends Cookie> cookiesIt = checkNotNull(cookies, "cookies").iterator();
+ if (!cookiesIt.hasNext()) {
return Collections.emptyList();
}
List<String> encoded = new ArrayList<String>();
- for (Cookie c : cookies) {
- if (c == null) {
- break;
- }
+ Cookie firstCookie = cookiesIt.next();
+ Map<String, Integer> nameToIndex = strict && cookiesIt.hasNext() ? new HashMap<String, Integer>() : null;
+ int i = 0;
+ encoded.add(encode(firstCookie));
+ boolean hasDupdName = nameToIndex != null ? nameToIndex.put(firstCookie.name(), i++) != null : false;
+ while (cookiesIt.hasNext()) {
+ Cookie c = cookiesIt.next();
encoded.add(encode(c));
+ if (nameToIndex != null) {
+ hasDupdName |= nameToIndex.put(c.name(), i++) != null;
+ }
}
- return encoded;
+ return hasDupdName ? dedup(encoded, nameToIndex) : encoded;
}
```

Figure 2. Source code change related to http header cookie encoding.

In this paper, we propose preprocessing a source code change through tokenization, camel case splitting, lowercase transformation, stemming, and stop word removal, in that order. Tokenization splits a sequence of words into word tokens with delimiters, which are typically white spaces. Camel case splitting splits a word that consists of two or more words, such as "nameToIndex" and "hasDupdName", into each separate word. Lowercase transformation converts uppercase characters in a word to lowercase characters. Stemming reduces a word to its base form. For example, "work", "working", and "worked" are all converted to "work" with stemming. Stop word removal removes specified stop words, keywords, and operators.

We implemented a source code change analyzer to perform the text processing using Apache Lucene 6.3.0 (https://lucene.apache.org/core/). The source code change analyzer uses WhitespaceTokenizer, WordDelimiterFilter, LowerCaseFilter, PorterStemFilter, and StopFilter. WhitespaceTokenizer splits a sequence of words into word tokens with whitespace delimiters. WordDelimiterFilter splits a word

token that consists of more than two words into its separate word tokens. LowerCaseFilter replaces uppercase characters in a word token with lowercase characters. PorterStemFilter reduces a word token into its base form. StopFilter removes English stop words and keywords of a programming language. We used the WordDelimiterFilter with WordDelimiterFilter.GENERATE_WORD_PARTS, WordDelimiterFilter.SPLIT_ON_CASE_CHANGE, and WordDelimiterFilter.SPLIT_ON_NUMERICS flags, and the StopFilter with the StopAnalyzer.ENGLISH_STOP_WORDS_SET option.

4.2.2. Extraction of Topics from Source Code Changes

LDA consists of training and inference phases. The training phase extracts topic distributions from given text documents. The inference phase infers topic distributions of new source code changes that are not included in the given text documents used in the training phase. The reviewer expertise generation and reviewer recommendation phases rely on LDA to extract the topic distributions of the source code changes. In this section, we briefly introduce the LDA topic extraction process with a collection of source code changes.

We define several notations by borrowing the basic LDA notations of [15] for describing the topic extraction process for source code changes. LDA requires a collection of source code changes, the number of sampling repetitions, a number of topics (K) to be extracted, and alpha and beta values as input. Given a collection of source code changes as $R = \{r_1, \ldots, r_n\}$, where r_i is a source code change, LDA first produces word-topic vectors z_{r_i} and then topic distribution vectors θ_{r_i} for each source code change in R. z_{r_i} represents the topic assignments of words in r_i. The topic assignment of each word in z_{r_i} is estimated to one of K topics through repetitive sampling. At the beginning of the sampling, the topic assignments of all the words in z_{r_i} are initialized randomly. At each sampling step, for each source code change r_i and each topic k, LDA estimates the probability of topic k being assigned to the j^{th} word in r_i. This is computed as follows:

$$p\left(r_{i,w_j} = k\right) = \frac{N_{r_i,-w_j+\alpha}}{N_{r_i} - 1 + W\alpha} \times \frac{N_{R,-w_j+\beta}}{N_R - 1 + K\beta} \quad (1)$$

where N_{r_i} is the total number of words in r_i. $N_{r_i,-w_j}$ is the number of words in r_i assigned to topic k, excluding the word w_j. N_R is the number of words assigned to topic k in all the source code changes in R. $N_{R,-w_j}$ is the number of times the word w_j is assigned to topic k in all source code changes in R, excluding the occurrence of the word w_j in r_i. W is the total number of words in R. α is the prior weight of a word in a topic. β is the prior weight of a topic in a source code change. LDA randomly assigns one of K topics to the word w_j according to the probabilities of the K topic computed by Equation (1) at each sampling process. At the end of the entire sampling process, LDA computes the topic distribution of each source code change θ_{r_i} based on the results of the topic assignments of the words in r_i. The topic distribution of a source code change is defined as follows:

$$\theta_{r_i} = \langle \varnothing_{z_1}, \ldots, \varnothing_{z_K} \rangle \quad (2)$$

The distribution value of each topic $\varnothing_z$ is computed as follows:

$$\varnothing_{z_k} = \frac{\text{\#. words assigned to the topic k in } r_i}{\text{\#. words in } r_i} \quad (3)$$

where z_k is a topic k and $\varnothing_{z_k}$ is the distribution value of the topic k, which ranges from 0 to 1. The sum of all the values of θ_{r_i} is 1.

4.2.3. Computation of the Review Expertise of the Developers

The last step of the review expertise generation phase is to compute the review expertise of the given developers. In this paper, we regarded that the review expertise of the developer is reflected by the review contribution of a developer to source code changes. At the beginning of the review expertise

generation phase, the past review history of the developers is given. The review history provides the review contributions of the developers to the source code changes. We define the review contribution of a developer as $R_d = \{r_1, \ldots, r_n\}$, where r_i is a source code change and d is the developer who reviewed the source code changes.

We propose here a method to compute the review expertise of a developer for topics of the source code changes based on the review contributions of the developer. As described in Section 4.2.2, a source code change has distribution values for K topics, as described by Equation (2). The review expertise of a developer for K topics is determined by the proportion of the source code changes reviewed by the developer. That is, for a specific topic, the review expertise of a developer is computed by dividing the sum of the distribution values of the topic in the source code changes reviewed by the developer with the sum of the distribution values of the topic in all reviewed source code changes R, as follows:

$$E_d(z) = \frac{\sum_{r_i \in R_d} \theta_{r_i}[z]}{\sum_{r_j \in R} \theta_{r_j}[z]} \tag{4}$$

where z is a topic, $\theta_{r_i}[z]$ is the distribution value of topic z in a source code change r_i. $E_d(z)$ indicates the cumulative review contribution of topic z of a developer in all the reviewed source code changes. $E_d(z)$ has a value between 0 and 1 and becomes larger as a developer reviews additional source code changes associated with topic z. Based on Equation (4), the overall review expertise of a developer on K topics is computed. We define the review expertise of a developer on K topics as a vector expression:

$$\text{RevExp}_d = \langle E_{z_1}, \ldots, E_{z_K} \rangle \tag{5}$$

where E_{z_i} is the abbreviation of $E_d(z_i)$ and RevExp_d represents the review expertise of a developer on K topics.

4.3. Reviewer Recommendation Phase

The reviewer recommendation phase consists of two steps: inferring topics of a new source code change and computing the developers' review scores. In the following subsections, we describe the steps in detail.

4.3.1. Inference of Topics of New Source Code Change

The reviewer recommendation phase takes a new source code change as input in order to recommend reviewers. Before extracting a topic distribution of a new source code change, the new source code change is also preprocessed as in the review expertise generation phase. Then, the preprocessed source code change and the topic model that has been generated in the training phase of the LDA are passed to LDA as input. The LDA inference process for a new source code change is the same to the LDA training process described in Section 4.2.2. LDA randomly assigns topics to the words in a new source code change based on the following equation through repetitive sampling:

$$p\left(r_{new,w_j} = k\right) = \frac{N_{r_{new},-w_j+\alpha}}{N_{r_{new}} - 1 + W\alpha} \times \frac{N_{R,-w_j+\beta}}{N_R - 1 + K\beta} \tag{6}$$

where r_{new} is a new source code change, $r_{new,\, w_j}$ is a word w_j in r_{new}, $N_{r_{new}}$ is the total number of words in r_{new}, $N_{r_{new},-w_j}$ is the number of words in r_{new} that are assigned to topic k, excluding the word w_j. The rest of the parameters have the same meaning as in Equation (1). After the entire sampling process, the topic distribution of a new source code change is finally determined based on Equations (2) and (3).

4.3.2. Computation of the Review Score

The final step of the reviewer recommendation phase is computing the review scores of the given developers for a new source code change. In this step, the review expertise of the developers

determined in the review expertise generation phase and the topic distribution of the new source code change inferred in the prior section are used. As a new source code change has a topic distribution for K topics, we believe that the developers who often reviewed the topics of the new source code change that have distribution value higher than 0 are more likely to be suitable as reviewers. The review score of a developer is computed as follows:

$$\text{RevScore}(d, \theta_{new}) = \sum_{z_i \in \theta_{new}} \text{RevExp}_d[z_i] \times \theta_{new}[z_i] \quad (7)$$

where θ_{new} is a topic distribution of a new source code change, $\text{RevScore}(d, \theta_{new})$ has a value of 0 or more and becomes larger as a developer has higher review expertise on the topics of a new source code change that have a distribution value higher than 0. The reviewer recommendation phase recommends the top-N developers in order of the high review scores of the developers computed with Equation (7).

Suppose that a topic distribution of a new source code change θ_{new} and review expertise of four developers for five topics is given as in Table 3. The new source code change only has distributions for topic 4 (0.8) and topic 5 (0.2). Thus, developers who have the review expertise for topic 4 and topic 5 are suitable to review the new source code change. For topic 4 and topic 5, the developers d_1, d_2, d_3, and d_4 have 0.9, 0.1, 0.3, and 0 and 0.6, 0.5, 0.3, and 0 of the review expertise, respectively. The developer d_4 should be excluded or has the lowest priority in the recommendation as d_4 does not have any review expertise for topic 4 and topic 5. On the other hand, it is reasonable to recommend developer d_1 as a reviewer with the highest priority as d_1 has much higher review expertise for both topic 4 and topic 5 than the developers d_2 and d_3. The developer d_3 should have higher priority than the developer d_2 as d_3 has higher review expertise for topic 4 despite having a little less review expertise for topic 5 than d_2. As a result, the developers are listed as reviewers in order of d_1, d_3, d_2, and d_4 based on their review scores.

Table 3. An example of a topic distribution of a new source code change and review expertise of four developers for five topics.

	T_1	T_2	T_3	T_4	T_5	**RevScore**
θ_{new}	0	0	0	0.8	0.2	-
d_1	0	0	0	0.9	0.6	0.8
d_2	0	0	0	0.1	0.5	0.2
d_3	0	0	0	0.3	0.3	0.3
d_4	0.8	0.6	0	0	0	0

4.4. Algorithm of the Proposed Approach

Figure 3 shows the algorithm of the proposed reviewer recommendation approach in pseudo code. The algorithm takes a set of source code changes reviewed, a set of developers, a number of K to be extracted, a number of sampling, the values of alpha and beta for LDA, a new source code change to be reviewed and a number of recommended developers as input, and then outputs a list of recommended reviewers. The body of the algorithm consists of two parts, the review expertise generation phase (from line 1 to line 17) and the reviewer recommendation phase (from line 18 to line 33). In the part of the review expertise generation phase, the reviewed source code changes given as an input are preprocessed (line 2). The preprocessing process is performed as described in Section 4.2.1. Then, the topic distributions of the reviewed source code changes are extracted using an LDA implementation with the values of K, S, α, β (line 3). After extracting the topic distributions of the source code changes, the review expertise for K topics are computed for each developer in the set of developers given as an input (from line 4 to line 17). For each topic k, the review contributions of a developer, $\text{ReviewContribution}_{d,k}$, is computed based on a set of source code changes reviewed by the developer (from line 9 to line 11) and the total topic distribution, $\text{TotalTopicProportion}_k$, is computed from all the source code changes (line 12 to line 14). Then, the review expertise of the developer

for topic k, $RevExp_{d,z_k}$, is determined by dividing his/her review contributions by the total topic distribution of topic k (line 15). In the part of the reviewer recommendation phase, the new source code change is preprocessed and then its topic distribution is inferred using an LDA implementation with the values of K, S, α and β (from line 19 to line 20). After that, for each developer and each topic k, the review scores of the developers are computed with the topic distribution of the new source code change and the computed review expertise of the developers (from line 22 to line 27). Finally, the developers are sorted by the computed review scores in descending order and top-N developers are selected from the ordered developers as the reviewer candidates (from line 30 to line 32).

Input :
R : A set of source code changes reviewed in past.
D : A set of developers (reviewer candidates)
K : A number of topics to be extracted
S : A number of sampling
α : A value of the alpha
β : A value of the beta
r_{new} : A new source code change to be reviewed
N : Top-N recommendations
Output :
Recommendations : A list of recommended reviewers.
Method :

1. // Review expertise generation phase
2. $R' \leftarrow preprocessing(R)$
3. $\theta \leftarrow TrainTopicsWithLDA(K, S, \alpha, \beta, R')$
4. for $d \in D$ do
5. $\quad R'_d \leftarrow reviewContribution(d, R')$
6. $\quad$ for $k \leftarrow 1$ to K do
7. $\quad\quad ReviewContribution_{d,k} \leftarrow 0$
8. $\quad\quad TotalTopicProportion_k \leftarrow 0$
9. $\quad\quad$ for $r'_i \in R'_d$ do
10. $\quad\quad\quad ReviewContribution_{d,k} \leftarrow ReviewContribution_k + \theta_{r'_i}[k]$
11. $\quad\quad$ end for
12. $\quad\quad$ for $r'_j \in R'$ do
13. $\quad\quad\quad TotalTopicProportion_k \leftarrow TotalTopicProportion_k + \theta_{r'_j}[k]$
14. $\quad\quad$ end for
15. $\quad\quad RevExp_{d,z_k} \leftarrow \frac{ReviewContribution_{d,k}}{TotalTopicProportion_k}$
16. $\quad$ end for
17. end for
18. // Reviewer recommendation phase
19. $r'_{new} \leftarrow preprocessing(r_{new})$
20. $\theta_{new} \leftarrow InferenceTopicsWithLDA(K, S, \alpha, \beta, r'_{new})$
21. $RevScore \leftarrow \emptyset$
22. for $d \in D$ do
23. $\quad$ for $k \leftarrow 1$ to K do
24. $\quad\quad RevScore_d \leftarrow RevScore_d + (RevExp_{d,z_k} \times \theta_{new}[z_k])$
25. $\quad$ end for
26. $\quad RevScore \leftarrow RevScore \cup RevScore_d$
27. end for
28. $DevList \leftarrow sortingDevelopersBy(RevScore)$
29. $Recommendations \leftarrow \emptyset$
30. for $i \leftarrow 1$ to N do
31. $\quad Recommendations \leftarrow Recommendations \cup DevList[i]$
32. end for
33. return Recommendations

Figure 3. Algorithm of the proposed approach.

5. Empirical Evaluation

We performed an empirical study to evaluate the proposed reviewer recommendation approach. In this section, we first introduce in Section 5.1 the subject projects used to perform the empirical study. In Section 5.2, we describe the collected data from the subject projects. In Section 5.3, we describe the implementation details of the proposed approach. We also introduce existing reviewer recommendation approaches used for comparison with the proposed approach in Section 5.4. In Section 5.5, we present the evaluation metric used for evaluating the reviewer recommendation performance. In Section 5.6, we report and discuss the results of the empirical study. Lastly, we present threats to the validity of this study in Section 5.7.

5.1. Subject Projects

To evaluate the effectiveness of the proposed reviewer recommendation approach, we consider open source projects on GitHub as our subject projects. As mentioned in Section 3, GitHub provides the pull-based development mechanism. Therefore, we can easily obtain various developers' review data from open source projects on GitHub. The review data is suitable for performing the reviewer recommendation experiments. In addition, GitHub also provides various software project development data. Hence, the data of GitHub has been widely used in previous studies [26–30].

We selected five open source projects: Bitcoin Core integration-staging tree (bitcoin), Ruby on Rails (https://github.com/rails/rails) (rails), KODI (https://github.com/xbmc/xbmc) (xbmc), and node-js (https://github.com/nodejs/node) (node), as the subject projects. bitcoin is an open source project that maintains and releases a bitcoin client software. rails is an open source project that uses an Model View Controller (MVC) pattern written in Ruby. xbmc is an open source project media player and entertainment for digital media. node is a JavaScript runtime built on Chrome's V8 JavaScript engine. tensorflow is an open source software library for high-performance numerical computation. As these projects are popular on GitHub and retain many contributors, there is an abundance of review data. Hence, we selected those projects as the subject projects for this study.

5.2. Data Collection

We collected the review data of the subject projects from the GHTorrent project. The GHTorrent project distributes project development data collected through the GitHub event stream as MySQL and MongoDB dump databases [23,31]. We downloaded a MySQL and a MongoDB dump database that contained development data from GitHub from December 2015 to October 2016 from the GHTorrent project website. We then collected pull requests submitted to the subject projects from 1 May 2016 to 31 October 2016. A pull request contains the author of the pull request, changed source file paths, and the files' source code changes and reviewers. We excluded the pull requests that did not contain any source file changes or that were missing reviewers because such pull requests could not be used as evaluation data. Table 4 shows the experimental datasets collected for each subject project. The columns are #. Pull requests, #. Commits, #. Reviewers, and #. Source file changes, which correspond to the number of pull requests, commits, reviewers, and changed source files, respectively.

Table 4. Experimental datasets.

Project	#. Pull Requests	#. Commits	#. Reviewers	#. Source File Changes
bitcoin	271	983	65	27949
rails	209	408	57	4521
xbmc	135	333	42	3584
node	573	1766	111	35117
tensorflow	157	359	132	6502

5.3. Implementation of Our Approach

The proposed reviewer recommendation approach requires several text processing techniques and an LDA implementation. The text processing techniques preprocess the given source code changes. An LDA implementation extracts the topic distributions of the preprocessed source code changes. We implemented the proposed reviewer recommendation approach with Apache Lucene Core 6.3 and MALLET. Apache Lucene Core is a text search engine library written in Java. We built a module of the text processing techniques, called source code change analyzer, using the Apache Lucene Core library. MALLET is a Java-based package for statistical natural language processing, document classification, clustering, topic modeling, information extraction, and other machine learning applications. MALLET provides a Gibbs sampling-based LDA. The proposed reviewer recommendation approach uses the LDA implementation of MALLET (http://mallet.cs.umass.edu/). For each subject project, we ran the LDA implementation with the K, alpha and beta of 20, 5.0, and 0.01, respectively.

5.4. Baseline Approaches

We evaluated the proposed reviewer recommendation approach by comparing it with REVFINDER [10] and cHRev [13]. REVFINDER is a reviewer recommendation approach based on file location similarity. By using four string comparison techniques, REVFINDER computes the review score of a developer by comparing the similarity of the paths of the requested files for review and the paths of files reviewed by the developer. cHRev is a reviewer recommendation approach based on the review count of developers to source files. The review score of a developer is computed based on how many times the developer reviewed the source files requested for review. The factors used for computing a review score of a developer in the proposed approach, REVFINDER, and cHRev are different. REVFINDER and cHRev consider the path similarity of the source files and review count of the source files as the factor for the computation, respectively. On the contrary, the proposed approach considers the topics of source code changes reviewed by the developers as the factor for review expertise computation. Therefore, we can investigate the effectiveness of the proposed approach by comparing the recommendation results of the proposed approach with the recommendation results of REVFINDER and cHRev.

5.5. Evaluation Metrics

To measure the effectiveness of the proposed reviewer recommendation approach, we used the top-N accuracy metric. The top-N accuracy metric has been widely used in evaluating recommendation systems [10,32]. The top-N accuracy of a reviewer recommendation approach is the proportion of the number of correct recommendation results against the total number of recommendations. Thus, the top-N accuracy of a reviewer recommendation approach is computed with the following equation:

$$\text{TopN Accuracy}(R) = \frac{\sum_{r \in R} \text{isCollect}(\text{ActualReviewers}_r,\ \text{TopN})}{|R|} \tag{8}$$

where R is a set of reviews to be recommended, r is a review, and ActualReviewer_r is the actual reviewer of r. TopN is the list of reviewers that is recommended by a reviewer recommendation approach. $\text{isCollect}(\text{ActualReviewers}_r,\ \text{TopN})$ has a value of 1 if TopN includes at least one reviewer involved in ActualReviewer_r; otherwise, it has a value of 0. According to [6,10,33], we chose the value of N to be 1, 3, 5, and 10.

5.6. Results & Discussion

In this subsection, we report the results of the empirical evaluation. For each subject project, we ran the proposed approach, cHRev, and REVFINDER. The three reviewer recommendation approaches all require a training dataset to compute the review scores of the developers. We split the collected review history data in each subject project into a training dataset and an evaluation dataset. We used

the review history data from 1 May 2016 to 31 June 2016 as a training dataset and data from 1 July 2016 to 31 October 2016 as the evaluation dataset. Table 5 shows the number of review data in the training data and the evaluation data.

Table 5. Number of review histories in the training and evaluation datasets from the subject projects.

Project	#. Train Set	#. Evaluation Set
bitcoin	636	347
rails	243	165
xbmc	163	170
node	1207	559
tensorflow	127	232

In this evaluation, we ran the proposed approach by setting K, alpha, and beta to 20, 5.0, and 0.01, respectively. The value of the parameter K is sensitive when analyzing topics of a software project because values that are too large or too small hinder exact topic generation [24]. Hence, according to [24], we set the value of K to 20. The values of alpha and beta were set as 5.0 and 0.01, which are default values in MALLET, respectively.

For each subject project, we evaluated the performances of the proposed approach, cHRev, and REVFINDER. Table 6 shows the results of the top-N accuracy of the approaches in the subject projects. For each subject project, when performing the top-10 recommendations, the proposed approach achieved 0.72, 0.71, 0.60, 0.75, and 0.39 of the top-10 accuracy, whereas cHRev and REVFINDER achieved 0.71, 0.58, 0.56, 0.56, and 0.31 and 0.65, 0.68, 0.62, 0.71, and 0.33 of the top-10 accuracy, respectively. The proposed approach obtained better top-10 accuracy than cHRev and REVFINDER, except for the *xbmc* project. On average, the proposed approach obtained 10% better top-10 accuracy than cHRev and obtained 4% better top-10 accuracy than REVFINDER.

Table 6. Results of top-N accuracy.

	Our Approach				cHRev				REVFINDER			
	Top-N				Top-N				Top-N			
Project	**1**	**3**	**5**	**10**	**1**	**3**	**5**	**10**	**1**	**3**	**5**	**10**
bitcoin	0.06	0.25	0.46	0.72	0.29	0.52	0.57	0.71	0.05	0.17	0.35	0.65
rails	0.22	0.53	0.61	0.71	0.22	0.43	0.48	0.58	0.34	0.56	0.63	0.68
xbmc	0.37	0.44	0.53	0.60	0.24	0.34	0.41	0.56	0.34	0.44	0.46	0.62
node	0.13	0.27	0.38	0.75	0.14	0.25	0.38	0.56	0.07	0.25	0.45	0.71
tensorflow	0.13	0.25	0.29	0.39	0.07	0.20	0.27	0.31	0.07	0.19	0.30	0.33
Avg.	0.18	0.35	0.45	0.64	0.19	0.35	0.42	0.54	0.17	0.32	0.44	0.60

Furthermore, we present the following null hypotheses to evaluate the improvement of the recommendation result of the proposed approach compared with cHRev and REVFINDER.

$H_{null,cHRev}$: There is no statistically significant difference between the results of the proposed approach and cHRev.

$H_{alternative,\ cHRev}$: There is a statistically significant difference between the results of the proposed approach and cHRev.

$H_{null,REVFINDER}$: There is no statistically significant difference between the results of the proposed approach and REVFINDER.

$H_{alternative,\ REVFINDER}$: There is a statistically significant difference between the results of the proposed approach and REVFINDER.

We used t.test function in R (https://www.r-project.org/) package to perform the student's *t*-test with the results of the top-10 accuracy of the recommendation approaches. Table 7 shows the *t*-values and *p*-values obtained with the Student's *t*-test. With 95% confidence, we reject $H_{null,\ cHRev}$ in rails,

node, and tensorflow and also reject $H_{\text{null, REVFINDER}}$ in bitcoin, node, and tensorflow. The results show that the proposed approach has improved performance compared to cHRev in rails, node, and tensorflow and compared to cHRev and REVFINDER in bitcoin, node, and tensorflow.

Table 7. Results of the paired student's *t*-test.

	cHRev			**REVFINDER**		
Project	***t***	***p***	**Decision**	***t***	***p***	**Decision**
bitcoin	1.8063	0.07181	Accept	4.7107	3.67×10^{-6}	Reject
rails	2.6461	0.008957	Reject	0.47026	0.6388	Accept
xbmc	1.4886	0.1385	Accept	0.18516	0.8533	Accept
node	10.122	2.20×10^{-16}	Reject	3.6361	0.000302	Reject
tensorflow	3.8933	1.30×10^{-4}	Reject	2.6808	7.87×10^{-3}	Reject

In the bitcoin project, there was extremely little difference in the top-10 accuracy between the proposed approach and cHRev. However, in the results of the top-3 accuracy, cHRev obtained the most superior performance and the proposed approach obtained the second best performance, followed by REVFINDER. The differences of the top-3 accuracy of the proposed approach and REVFIDNER with cHRev are 27% and 37%, respectively. To understand why cHRev could obtain the most superior performance, we manually investigated the recommendation results in the bitcoin project. For the recommendation results, we compared the top-3 lists of cHRev and the proposed approach and confirmed that actual reviewers who have extremely little review history were involved only in the top-3 lists of cHRev. In general, the developers who have much more review history should be recommended more than those having less review history because the developers have reviewing experiences in multiple source files. The proposed approach and REVFINDER are basically designed to assign a high review score to such developers. Therefore, the developers who have little review history are mostly located with low ranks on the recommendation list by the proposed approach and REVFINDER. This caused the proposed approach and REVFINDER to have lower top-5 recommendation accuracy than cHRev. However, the proposed approach and REVFINDER outperformed cHRev in other projects.

Overall, the proposed approach showed improved performance compared to REVFINDER in the subject projects. To check out the advantage of the proposed approach in reviewer recommendation and the reason that the proposed approach could obtain better recommendation performance than REVFINDER, we also investigated the recommendation results of the proposed approach and REVFINDER in the bitcoin project. We found that REVFINDER had a limitation in recommending the correct reviewers for the source file "src/main.cpp". In the bitcoin project, the source file is one of the most often changed and reviewed source files. The source file was changed 134 times and reviewed by 40 developers from 1 May 2016 to 31 June 2016. It is difficult to select some appropriate reviewers among the developers with only their review expertise of the file path similarity or review count because most of the developers have a similar review experience for the source file. On the other hand, the proposed approach can complement such a limitation by considering the review expertise of the developers for the topic distribution of the source code change of the source file. In the actually performed recommendation by the proposed approach for the commit fe6f4056bb1e2be4b0fd84bb1752141b8bef38, which contained the source code change of the source file "src/main.cpp", an actual reviewer to the commit was precisely recommended within the top-5 according to his review expertise on the topic distribution of the source code change of the source file. The recommended reviewer indeed had high review expertise of the dominant topic of the source code change of the source file "src/main.cpp".

In this empirical study, we showed that it is effective to consider the information of topics of source code changes reviewed by developers in reviewer recommendation. The results showed that the file path similarity and review count based reviewer recommendation approaches have limitations

in recommending some reviewers. We believe that the proposed approach can complement the limitations of cHRev and REVFINDER and can improve the code review process of a software project.

5.7. Threat to Validity

The internal threats to the validity of this study are concerned with the result of the topic extraction of LDA. The computation of the review expertise of our reviewer recommendation approach depends substantially on LDA. Review expertise is computed based on the topic distributions of source code changes. An LDA implementation is necessary to extract the topic distributions of source code changes. The result of the topic extraction of LDA is affected by the number of source code changes used as input [15]. When an extremely small number of source code changes are used as an input dataset for LDA, the quality of the topic extraction of the source code changes will decrease. Hence, we cannot ensure the application of our reviewer recommendation approach in the initial development phase of a software project. However, the existing reviewer recommendation approaches REVFINDER and cHRev also have the same limitations.

The external threats to the validity of this study are concerned with the generalization of the result of the empirical evaluation study. In this study, we used only five open source projects experimental projects. The result of the study may not be generalizable to other open source projects. To reduce the bias, we gave much effort to choosing the experiment projects. We selected the experiment projects to differ in language, domain, and scale. Thus, we expect that results similar to those in this study can be obtained in other projects that have identical language, similar domain and scale, and a similar strategy of the review process.

6. Related Work

In software project development, the reviewer assignment problem is a significant matter. Thongtanunam et al. performed an exploratory study on the impact of the reviewer assignment problem [10]. They investigated the impact on the reviewing time of the reviews with reviewer assignment problem in four open source projects: Android, OpenStack, Qt, and LibreOffice. It was found that 4–30% of the reviews in the open source projects had a reviewer assignment problem. Furthermore, they found that, on average, the reviews with the reviewer assignment problem required an extra 12 days to complete the review process. Reviewer recommendation is beneficial to resolving the reviewer assignment problem. REVFINDER, which is a file location-based reviewer recommendation approach, was presented by Thongtanunam et al. REVFINDER recommends reviewers who mainly review files located in paths similar to the files requested for review. They showed the effectiveness of REVFINDER in the open source projects. Zanjani et al. proposed cHRev, a reviewer recommendation approach based on review count [13]. cHRev considers how many times the developer reviewed the source files requested for review as a factor in determining the review expertise of developers relative to the source files. In an experiment on three open source projects and a commercial software project, they showed the effectiveness of cHRev. In this study, we proposed an LDA-based reviewer recommendation approach. The proposed approach extracts topic distributions of reviewed source code changes using LDA and then computes the review expertise of developers for the extracted topics. This is a significant difference between our study and the previous studies.

7. Conclusions

After a source code is changed, developers must review the changed source code to integrate it into a software project. The reviewer assignment problem delays the code review process of a software project. Hence, developers should rapidly and accurately assign appropriate reviewers to newly submitted source code changes. In this paper, we proposed a reviewer recommendation approach based on LDA. The proposed reviewer recommendation approach consists of the review expertise generation and reviewer recommendation phases. The review expertise generation phase generates the review expertise of developers for topics of source code changes from the past review

history of a software project. The reviewer recommendation phase computes the review scores of developers according to the topic distribution of a given source code change and the review expertise of developers. In an empirical study on five open source projects, the proposed approach obtained better recommendation accuracy than the existing reviewer recommendation approaches.

In future work, we will conduct more experiments on various projects to reduce the external threats of generalization of our approach. Additionally, we will explore a way to improve reviewer recommendation performance. For instance, we will study how much an approach that combines our approach and other existing reviewer recommendation approaches can improve reviewer recommendation performance.

Acknowledgments: This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (2016R1D1A1B03931276). This study was also supported by the BK21 Plus Project (SW Human Resource Development Program for Supporting Smart Life), funded by the Ministry of Education, School of Computer Science and Engineering, Kyungpook National University, Korea (21A20131600005).

Author Contributions: Jungil Kim collected all of the experimental data, performed the experiment, analyzed the result of the experiments, and wrote the paper. Eunjoo Lee supervised this study, suggested the research approach, and designed the overall experiment method.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Jeong, G.; Kim, S.; Zimmermann, T.; Yi, K. Improving Code Review by Predicting Reviewers and Acceptance of Patches. Available online: https://pdfs.semanticscholar.org/2c9b/1dac31694c65466487dd11534d41cf328d32.pdf (accessed on 20 March 2018).
2. Fejzer, M.; Przymus, P.; Stencel, K. Profile based recommendation of code reviewers. *J. Intell. Inf. Syst.* **2017**, 1–23. [CrossRef]
3. Ciolkowski, M.; Laitenberger, O.; Rombach, D.; Shull, F.; Perry, D. Software inspections, reviews & walkthroughs. In Proceedings of the 24th International Conference on Software Engineering, Orlando, FL, USA, 19–25 May 2002; pp. 641–642.
4. Bernhart, M.; Mauczka, A.; Grechenig, T. Adopting code reviews for agile software development. In Proceedings of the AGILE Conference, Nashville, TN, USA, 9–13 August 2010; pp. 44–47.
5. Bosu, A.; Carver, J.C. Peer code review in open source communities using reviewboard. In Proceedings of the ACM 4th Annual Workshop on Evaluation and Usability of Programming Languages and Tools, Tucson, AZ, USA, 21 October 2012; pp. 17–24.
6. Balachandran, V. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In Proceedings of the 35th International Conference on Software Engineering, San Francisco, CA, USA, 18–26 May 2013; pp. 931–940.
7. Beller, M.; Bacchelli, A.; Zaidman, A.; Juergens, E. Modern code reviews in open-source projects: Which problems do they fix? In Proceedings of the 11th Working Conference on Mining Software Repositories, Hyderabad, India, 31 May–1 June 2014; pp. 202–211.
8. Yu, Y.; Wang, H.; Yin, G.; Ling, C.X. Who should review this pull-request: Reviewer recommendation to expedite crowd collaboration. In Proceedings of the 21st Asia-Pacific Software Engineering Conference, Jeju, South Korea, 1–4 December 2014; pp. 335–342.
9. Yu, Y.; Wang, H.; Yin, G.; Wang, T. Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment? *Inf. Softw. Technol.* **2016**, *74*, 204–218. [CrossRef]
10. Thongtanunam, P.; Tantithamthavorn, C.; Kula, R.G.; Yoshida, N.; Iida, H.; Matsumoto, K. Who should review my code? A file location-based code-reviewer recommendation approach for modern code review. In Proceedings of the IEEE 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER), Montreal, QC, Canada, 2–6 March 2015; pp. 141–150.
11. Bacchelli, A.; Bird, C. Expectations, outcomes, and challenges of modern code review. In Proceedings of the 2013 International Conference on Software Engineering, San Francisco, CA, USA, 18–26 May 2013; pp. 712–721.

12. Xia, X.; Lo, D.; Wang, X.; Yang, X. Who should review this change: Putting text and file location analyses together for more accurate recommendations. In Proceedings of the IEEE International Conference on Software Maintenance and Evolution, Bremen, Germany, 29 September–1 October 2015; pp. 261–270.
13. Zanjani, M.B.; Kagdi, H.; Bird, C. Automatically recommending peer reviewers in modern code review. *IEEE Trans. Softw. Eng.* **2017**, *42*, 530–543. [CrossRef]
14. Wang, Z.; Perry, D.E.; Xu, X. Characterizing Individualized Coding Contributions of OSS Developers from Topic Perspective. *Int. J. Softw. Eng. Knowl. Eng.* **2016**, *27*, 91–124. [CrossRef]
15. Blei, D.M.; Ng, A.Y.; Jordan, M.I. Latent dirichlet allocation. *J. Mach. Learn. Res.* **2003**, *3*, 993–1022.
16. Nguyen, A.T.; Nguyen, T.T.; Al-Kofahi, J.; Nguyen, H.V.; Nguyen, T.N. A topic-based approach for narrowing the search space of buggy files from a bug report. In Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering, Lawrence, KS, USA, 6–10 November 2011; pp. 263–272.
17. Xie, X.; Zhang, W.; Yang, Y.; Wang, Q. DRETOM: Developer recommendation based on topic models for bug resolution. In Proceedings of the 8th International Conference on Predictive Models in Software Engineering, Lund, Sweden, 21–22 September 2012; pp. 19–28.
18. Chen, T.H.; Shang, W.; Nagappan, M.; Hassan, A.E.; Thomas, S.W. Topic-based software defect explanation. *J. Syst. Softw.* **2017**, *129*, 79–106. [CrossRef]
19. Hannebauer, C.; Patalas, M.; Stünkel, S.; Gruhn, V. Automatically recommending code reviewers based on their expertise: An empirical comparison. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, Singapore, 3–7 September 2016; pp. 99–110.
20. Steyvers, M.; Griffiths, T. Probabilistic topic models. In *Handbook of Latent Semantic Analysis*; Association for Computing Machinery (ACM): New York, NY, USA, 2007; Volume 55, pp. 77–84. [CrossRef]
21. Deerwester, S.; Dumais, S.T.; Furnas, G.W.; Landauer, T.K. Indexing by latent semantic analysis. *J. Am. Soc. Inf. Sci.* **1990**, *41*, 391–407. [CrossRef]
22. Hofmann, T. Probabilistic latent semantic indexing. In Newsletter of ACM SIGIR Forum, Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Berkeley, CA, USA, 15–19 August 1999; Volume 51, pp. 50–57.
23. Gousios, G.; Spinellis, D. GHTorrent: Github's data from a firehose. In Proceedings of the 9th Working Conference on Mining Software Repositories, Zurich, Switzerland, 2–3 June 2012; pp. 12–21.
24. Hindle, A.; Godfrey, W.M.; Holt, C.R. What's hot and what's not: Windowed developer topic analysis. In Proceedings of the IEEE International Conference on Software Maintenance, Edmonton, AB, Canada, 20–26 September 2009; pp. 339–348.
25. Lukins, S.K.; Kraft, N.A.; Etzkorn, L.H. Source Code Retrieval for Bug Localization Using Latent Dirichlet Allocation. In Proceedings of the 15th Working Conference on Reverse Engineering, Antwerp, Belgium, 15–18 October 2008; pp. 155–164.
26. Gousios, G.; Pinzger, M.; Deursen, A. An exploratory study of the pull-based software development model. In Proceedings of the 36th International Conference on Software Engineering, Hyderabad, India, 31 May–7 June 2014; pp. 345–355.
27. Gousios, G.; Zaidman, A.; Storey, M.A.; Deursen, A. Work practices and challenges in pull-based development: The integrator's perspective. In Proceedings of the International Conference on Software Engineering, Florence, Italy, 16–24 May 2015; pp. 358–368.
28. Zhang, L.; Zou, Y.; Xie, B.; Zhu, Z. Recommending Relevant Projects via User Behaviour: An Exploratory Study on Github. In Proceedings of the 1st International Workshop on Crowd-Based Software Development Methods and Technologies, Hong Kong, China, 17 November 2014; pp. 25–30.
29. Constantinou, E.; Mens, T. Socio-technical evolution of the Ruby ecosystem in GitHub. In Proceedings of the IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, Klagenfurt, Austria, 20–24 February 2017; pp. 34–44.
30. Blincoe, K.; Harrison, F.; Damian, D. Ecosystems in GitHub and a Method for Ecosystem Identification using Reference Coupling. In Proceedings of the 12th Working Conference on Mining Software Repositories, Florence, Italy, 16–24 May 2015; pp. 202–207.
31. Gousios, G.; Vasilescu, B.; Serebrenik, A.; Zaidman, A. Lean GHTorrent: GitHub data on demand. In Proceedings of the 11th Working Conference on Mining Software Repositories, Hyderabad, India, 31 May–1 June 2014; pp. 384–387.

32. Tian, Y.; Wijedasa, D.; Lo, D.; Le Goues, C. Learning to Rank for Bug Report Assignee Recommendation. In Proceedings of the IEEE 24th International Conference on Program Comprehension, Austin, TX, USA, 16–17 May 2016; pp. 1–10.
33. Thongtanunam, P.; Kula, R.G.; Cruz, A.E.C.; Yoshida, N.; Iida, H. Improving code review effectiveness through reviewer recommendations. In Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering, Hyderabad, India, 2–3 June 2014; pp. 119–122.