Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

Vocabulary and Time Based Bug Assignment (*VTBA*): A Recommender System for Open-source Projects

Ali Sajedi-Badashian* | **Eleni Stroulia**

Department of Computing Science, University of Alberta, AB, Canada

Correspondence
*Ali Sajedi-Badashian, 2-09 Athabasca Hall, Edmonton, AB, Canada. T6G 2E8. Email: alisajedi@ualberta.ca

Funding information
Alberta Innovates - Technology Futures (AITF) and Queen Elizabeth II Scholarships

Summary

Bug-assignment (BA), the task of ranking developers in terms of the relevance of their expertise to fix a new bug report is time consuming, which is why substantial attention has been paid to developing methods for automating it.

In this paper, we describe a new bug-assignment approach that relies on two key intuitions. Similar to traditional bug-assignment methods, our method constructs the expertise profile of project developers, based on the textual elements of the bugs they have fixed in the past; unlike traditional methods, however, our method considers only the programming keywords in these bug descriptions, relying on Stack Overflow as the vocabulary for these keywords. The second key intuition of our method is that recent expertise is more relevant than past expertise, which is why our method weighs the relevance of a developer's expertise based on how recently they have fixed a bug with keywords similar to the bug at hand.

We evaluated our BA method using a data set of 93k bug-report assignments from thirteen popular GitHub projects. In spite of its simplicity, our method predicts the assignee with high accuracy, outperforming state-of-the-art methods.

KEYWORDS:
bug-assignment, software engineering, GitHub, information retrieval, *TF-IDF*

1 | INTRODUCTION

As a key task of the software quality-assurance process, bug-assignment (BA) aims at identifying the most appropriate developer(s) to fix a given bug. Typically, BA methods consider the developers' previous development activities as indicators of their expertise and rank the developers' relevance to the bug in question using a variety of heuristics. BA is an important problem for the software-engineering industry, and it involves a number of challenging questions, including what project data to consider as evidence for a developer's expertise, and how to utilize developers' expertise to recommend the best developers to fix new bugs.

Because of its importance, the problem has already received substantial attention over the past decade [1,2,3,4,5,6,7]. Nevertheless, BA is still a time-consuming task in software development [8,9]. Large projects receive hundreds of bug reports daily [10], which get recorded in their issue-tracking tools, which typically support flexible searching and, in some cases, duplicate detection. No issue-tracking tool, however, automates the BA task, which is still manual or, at best, semi-automated. Especially for big projects, a valid and accurate BA tool would be extremely desirable [11,12]. In big open-source projects, automating the BA process is part of an arrangement called Recommendation Systems for Software Engineering (RSSE) [13]. RSSE remarkably saves project managers' time and effort since it cuts down their every days's recurring tasks.

The first key question in developing an RSSE for assigning a bug to the developer best qualified to fix it is to decide on "how to match the information available on the bug with the information available on the developer's prior experience and contributions". The methodological assumption of our work is that *technical terms* and their *time of usage* are critical in this "matching". Even though much of the information about bugs and

developers is textual, recognizing expertise relevant to a bug is very different from textual-similarity assessment, which has been the prevalent paradigm for this task to date. This is why some previous research used the traditional *TF-IDF*[14] to differentiate between specific and common keywords[11,12,15,16]. Nonetheless, to the best of our knowledge, only one of them (Shokripour *et al.*)[15] used a time-based variant of the original *TF-IDF* method as the main similarity metric between bug reports and developers. Nevertheless, it used the document-length and frequency of the keywords (as the requirements of the *TF-IDF* method), plus the "last" time of usage of the keywords. However, it ignores other usages of the keywords and their times.

In this paper, we propose a new similarity metric originally based on *TF-IDF*, but with two important enhancements. 1) The metric deals with the relevance of a developer's expertise to a given bug by considering the **technical-keyword space**. We ignore all words that do not belong to the technical vocabulary of Stack Overflow tags, which are curated by the software-engineering community. Our method weighs the importance of the keywords based on their distinctiveness in Stack Overflow. 2) Furthermore, the developers' expertise shifts as their tasks evolve over time. So, our similarity metric takes into account the recency of the technical-keyword appearance in the developer's record. It benefits **high granularity of the time** of usage of the terms in previous BAs. In effect, our similarity metric is a **v**ocabulary and **t**ime-aware **b**ug-**a**ssignment, henceforth VTBA. We show that our model notably enhances the assignee recommendation accuracy.

To evaluate this metric and to examine the relative importance of its two constituent intuitions, we have curated an extensive data set, including bug reports from thirteen open-source projects, their meta-data and textual information and their assignee(s). This data set contains 93k bug-assignments and we will publish it, as well as all the source code and instructions for our experiment, for further research and investigations. Using this data set, we demonstrate that VTBA outperforms current state-of-the-art methods.

The remainder of this paper is organized as follows. Section 2 places our work in the context of the recent relevant literature. Section 3 describes the new metric we have developed for estimating the relevance of a developer's expertise for a given bug. Sections 4 and 5 describe our data set and details of the experiment we performed. Sections 6 and 7 report our results and discuss their implications. Finally, Section 8 concludes with a review of the lessons we learned from this work and outlines some avenues for future research.

2 | BACKGROUND AND RELATED RESEARCH

The most prevalent formulation of the BA task is as follows: "Given a new bug report, identify a ranked list of developers, whose expertise (based on their record of contributions to the project) qualifies them to fix the bug"[2,17,18,15]; this is the formulation we try to solve by our BA method -ignoring other formulations like "team BA"[19,20] and "multi-objective BA"[21,22,23,7].

We surveyed the BA research and introduced a selection including 13 studies (based on their reproducibility criteria, size of their developer community and number of bug reports in their experiments, etc.), representative of BA research[24,25]. Table 1 outlines those studies[1], covering various approaches and methods towards BA, ordered by publication date.

TABLE 1 A review of selected studies about BA, in chronological order

Method	A summary of the techniques used
Čubranić and Murphy[28]	It uses a **Naïve Bayes** classifier to assign each bug report (a "text document" consisting of the bug summary and description) to a developer (seen as a topic category or the "class") who actually fixed the bug. When a new bug report arrives, it uses the textual fields of the bug to predict the related class (e.g., developer).
Canfora and Cerulo[29]	It uses an **Information Retrieval (IR)** approach; It assumes that the developers who have solved similar bug reports in the past are the best candidates to solve the new one. So, it considers each developer as a document by aggregating the textual descriptions of the previous change requests that the developer has addressed. Given a new bug report, uses a probabilistic IR model and considers its textual description as a query to retrieve a candidate from the document (developer) repository.
Jeong, *et al.*[1]	It captures tossing probabilities between developers from tossing history of the bugs. Then makes a **tossing graph** of developers based on Markov Model. In this graph, the nodes are developers and the weight of the directed edges show the probability of tossing from one developer to the other. Finally, for predicting the assignees of a bug, it first produces a list of developers using a Machine Learning (ML) method. Then, after each developer in this list, adds the neighbor developer with the most probable tossing weight from the graph.

continued ...

[1]Three studies are conference version of other studies by the same authors;[2],[26] and[27]. So, we merged them into their journal version in the table.

TABLE 1 A review of selected studies about BA, in chronological order

… continued

Method	A summary of the techniques used
Matter *et al.* [17]	It employs the **Vector Space Model (VSM)**; Considers the source code contributions and the previous bug fixing as evidence of expertise and builds a vocabulary of "technical terms". It builds this vocabulary from the technical terms the developer used in the source code (captured by diff of the submitted revision) or commit messages. Also adds to this vocabulary the technical terms mentioned in the previous bug reports assigned to the developer. As a result, the developer's expertise is modeled and captured as a term vector. Given a new bug report, it calculates the **cosine distance** between the new bug report's term vector and the developers', and sorts them based on this score and reports the top ones.
Tamrawi *et al.* [26,30]	It introduces a **fuzzy** approach toward BA; it computes a score for each "developer - technical term" based on the technical terms available in previous bug reports and their fixing history by the developers. Considering a new bug report, it calculates a score for each developer as a candidate assignee by combining his/her scores for all the technical terms associated with the bug report in question. Then sorts the developers based on this score. The newer version, [30], also applies a term selection method to reduce noise data and speed up the algorithm. It extracts the top k terms that are most related with each developer. Then, when calculating the fuzzy score for each developer, just considers those selected terms and ignores other terms.
Bhattacharya and Neamtiu [2] Bhattacharya *et al.* [31]	It is similar to [1], but improves the **tossing graph** by adding labels for each edge. The label of each edge indicates product, component and latest activity date. Then, it recommends three developers based on an ML method. Also, the tosee ranking uses the graph labels (tossing probability, product, component and last activity date of the developer) for each of the top two developers in this list to recommend a substitute (tosee) and enhance this list to a top-5 recommendation.
Shokripour *et al.* [5]	It uses **bug report localization**, **Information Extraction (IE)** and **Natural Language Processing (NLP)**; it first applies IE and NLP techniques on file-related components (e.g.„ commit messages and their comments, plus their related source code elements including phrases in methods and classes) and also the bug reports and elicits their important phrases. When a new bug arrives, it estimates the location of the new bug (i.e.„ the files that should be changed to fix the bug) by comparing the important phrases in the bug report and other file-related components as mentioned above. Then, recommends the developers with the most activity regarding those files –according to the historical data.
Zhang *et al.* [11]	It builds a **heterogeneous social network** of developers, bugs and their comments, components and products. Then, selects the top *k* similar bug reports to the given new bug report by using **KNN** classification (a machine learning method that finds k closest instances to the given instance in the feature space), **Cosine similarity** and **tf-idf**. After that, extracts the list of commenters of those k bug reports as the narrowed list of "candidate developers" and obtains the score of each developer by calculating overall **heterogeneous proximity** of each developer with all other "candidate developers" on component and product of the new bug report, using the previously built network.
Cavalcanti *et al.* [27] Cavalcanti *et al.* [32]	It introduces a semi-automatic approach that combines rule-based expert system (RBES) with **Information Retrieval (IR)** methods. First, summarizes the previous bug reports and extracts some simple rules based on meta-data (e.g., component of a bug, or being critical) or keywords in the bug report, and the real assignee. These rules can decide on some circumstances which developer should be assigned to a new bug. In the second phase, if the simple rules cannot recommend any developer for the new bug report, uses the machine learning **SVM** classifier to consider previous assignments to developers and assign a label (developer) to it (e.g., the developers who fixed similar bug reports are most likely to fix the new one).
Sun *et al.* [33]	First, it analyzes and extracts the commits related to the issue request. This is done by obtaining **cosine** similarity between new bug report and historical commits. Those commits are considered relevant commits and the changed source code is considered relevant source code. Authors of those commits are considered as candidate developers to fix the new issue. Finally it uses **Collaborative Topic Modeling (CTM)** to give a score to each of those developers (based on shared keywords in their relevant source code and the new issue) and sort and recommend them to fix the new issue.

As shown in this table, previous research in BA approached the problem in different manners. Older approaches used Machine Learning (ML) techniques that mostly tried to predict the next assignees by investigating the "relations" between the developers and the previously fixed bug reports and their technical terms [28] (although some of the newer approaches used ML approaches and combined them with other methods). The performance of these approaches depends on the number of previously fixed bugs [5]. So, ML-based methods may perform poorly in small projects (e.g., poor results for GCC project in [34]) [5]. Then IR-based similarity metrics [29,17] and fuzzy and statistical approaches [26,30] appeared. These methods utilized an expertise perspective for the developers regarding keywords. Based on which developer fixed which bug, they gathered an expertise profile for the developers to make a decision upon arrival of new bug reports.

Recent studies mostly focused on IR based activity profiling since it usually leads to higher accuracies [35]. While some of those approaches mostly rely on textual information (e.g., title and description) of the –new and old– bug reports as clues for indication of expertise of developers and matching with new bug reports [28,26], others used variety of meta-data fields (e.g., component, product, severity and operating system) [17] to address some of the limitations. But still text-based methods are the most effective techniques used [15,36].

In recent years, some of the studies combined two or more different methods (e.g., ML, tossing graphs, Information Extraction, NLP and IR) [1,31,2,5,11,32,27,33]. Some of these studies [11] showed a tendency towards a *social* point of view, combined with the other methods. Some other examples (not shown in the above table) are building a social network of developers to model their relationship with each others or with bugs or even source code components [18,12] or combining KNN and IR methods [37].

Conceptually closer to our work are BA methods that consider the timeline of developers' activities and the importance of keywords:

First, previous research considered the *time* of previous evidence of expertise of developers as indication of relevance to the new bug report. The old evidence are outdated and have less effect on appropriateness of the developer for the new tasks. Having a new bug report at hand, they considered the time of "last" usage of its keywords by a developer, to obtain the recency factor of that keyword for the mentioned developer. Then, they apply this factor in the developer's score [10,15]; the former considered *last usage time* in a custom expertise formula and the later embedded it in the *TF-IDF* formula.

However, we believe that a fine-grained view over the usage of "all" the keywords by developers is needed; in fact, higher granularity over *time*, helps providing a more precise weighting for all the times a keyword has been used by a developer and gain more practical scoring scheme. Our similarity metric considers all the times of the usage of a term by a developer.

Second, the *importance* of the keywords is captured by some previous research. A number of previous BA studies used *TF-IDF* as a term-weighting technique [14] to emphasize on some keywords [12,11]. The motivating assumptions behind such term weighting are that some keywords are more important than the others because they are more specific and/or distinctive, or because they better capture specific interests of developers. There are other studies that tried to deal with different words differently. Aljarah *et al.* [3] uses Log-Odds-Ratio as a term-selection technique to identify the discriminating terms (i.e., those that are frequently used in the bugs assigned to a developer, but not frequently used by the others) and removed the rest of the terms. This way, it emphasizes on specific keywords that each developer uses or responds explicitly –by fixing the bugs containing those keywords.

The problem with these methods is that they just use the frequencies of the terms in the corpus, to determine importance of the keywords. It is true that less-frequently used keywords might be more important to focus (rather than widely used, general-purpose keywords), but there is no indication of the "technicality" of those keywords. We believe that capturing *less-frequently used "technical"* terms can help obtain the similarity of a bug report to a developer more precisely. Our approach uses Stack Overflow for inferencing about the technicality of the terms and obtaining term-weights consequently. Stack Overflow, as the leading question answering platform, is the first choice of most developers in the world, for seeking their programming answers [38,39]. With more than 15 million questions as of now, it covers every important programming topic [39,40]. In our previous research, we proposed a model to recommend developers in GitHub solely based on their posted questions and answers in Stack Overflow [41]. We used neither term-weights nor GitHub contributions in that study. Also, unlike our previous research, in the current study, we do not use information of questions and answers. In fact, the neat structure of the tags and their usage in more than 15 million questions inspired us to use them as a rich set of software-programming keywords. So, in this study, we use the set of Stack Overflow tags as a vocabulary to obtain general weights for the technical terms[2]. The usage of Stack Overflow tags is based on the general idea of "controlled vocabulary", a concept in IR that helps organize the domain knowledge of a problem for subsequent processing and retrieval [42].

We add these two factors (*time of usage* and *importance* of the keywords) to the well-known *TF-IDF* metric, and obtain a fine-grained method, called VTBA.

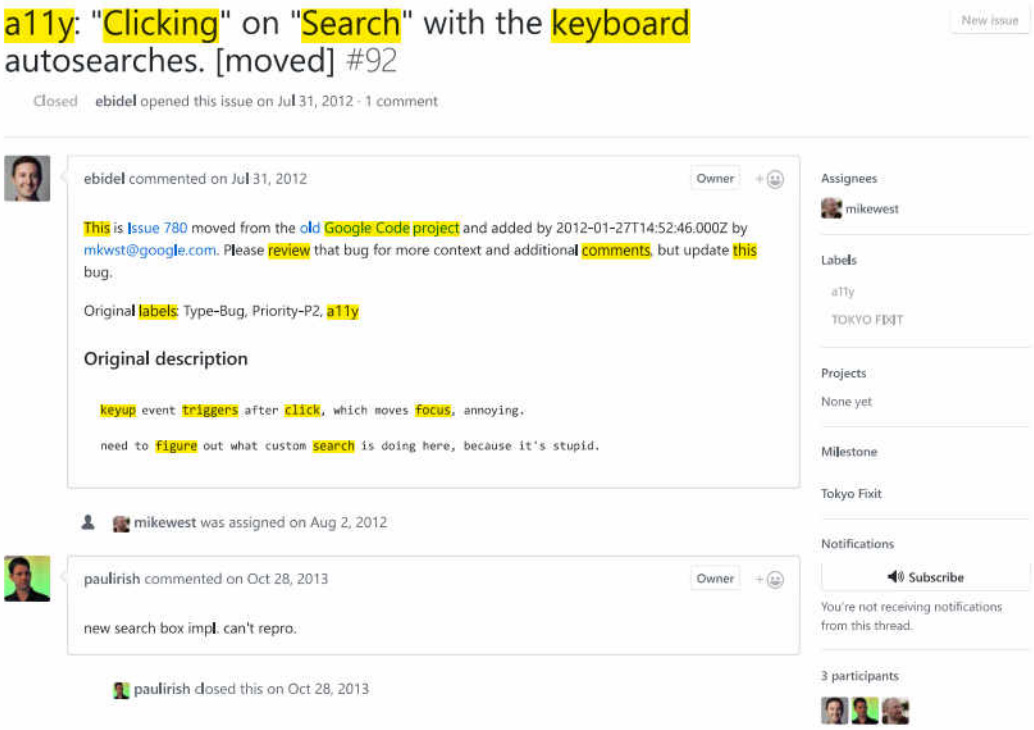
3 | RECOGNIZING DEVELOPERS WITH RELEVANT EXPERTISE: THE *VTBA* METRIC

In the IR formulation of BA, the profiles of developers (including the text of previously fixed bugs and other contributions) correspond to the *documents* and the description of the new bug report is considered as the *query*. Using this notation, the only approach combining *TF-IDF* (as the main scoring function for developers regarding a new bug report) with *time* was proposed by Shokripour *et al.* [15 3]. This approach added a recency factor based on the **last time of usage** of keywords by the developers. It only considers the time of last usage of the terms, but we believe it is needed to consider **all the usage times of the keywords** since the usage of the terms is scattered over time. In fact, more granularity is needed to capture the time of usage of a keyword by a developer precisely. Second, most of the previous methods infer the term weights based on raw

[2]Note that this is the only usage of Stack Overflow in the current study. There is no overlap with our previous work.

[3]Here, the usage of *TF-IDF* as the main scoring function regarding a developer (document) for a new bug (query) is regarded. There are a few other studies that used *TF-IDF* for simple term-weighting [11,32,12] within another main method and are excluded here.

FIGURE 1 A sample bug report (#92) in *www.html5rocks.com*



counts of how many times they appear in the project. They do not consider which of these terms are part of the *technical*-keywords of software-programming vocabulary, which is likely to be more important than regular words in English. Third, weights inferred from small projects (or at the early stages of bigger projects) are bound to be inaccurate, since there are not enough documents available. Finally, due to the dynamic nature of term weights calculated over the history of the project lifecycle and/or the developers' contributions, most previous methods need to re-calculate the term weights while processing new bug reports. Although this is mitigated by exploiting inverted indices, it still can be time-consuming since some bug reports include a considerable number of generic keywords.

To mediate these shortcomings, it is needed to use a *term-weighting* scheme that (a) has the ability to consider the timestamps of the (sub-)documents that contain these keywords; (b) emphasizes specific, *technical* (programming-related) keywords rather than general ones; (c) is independent of the corpus or less dependent so that less re-calculations are needed during time; and (d) reduces the search space by filtering the text of bug reports and making them shorter.

To set the context for a better understanding of the dynamics of bugs and their assignment, we use the example in Figure 1. The figure depicts a sample bug report[4] in one of the projects[5] we studied in our experiments. On the page of each bug, GitHub shows the title, description, meta-data elements (reporting date, reporter, labels, etc.), and the work around that bug. The bug report depicted in this example is reported on Jul 31th, 2012 and closed on Oct 28th, 2013. During this time period, the interactions around the bug –comments, being assigned or closed– are shown in the page of the bug report. If the bug is referenced from a commit or other actions, then an entry will be shown in the interactions section (bottom section of the bug). In this figure, the keywords in the bug's title and description that are also a Stack Overflow tag are highlighted.

A developer who fixed a bug report, is considered knowledgeable regarding the keywords mentioned in that bug report (e.g., title and description). This fact is usually taken into account by BA approaches. However, some of these keywords are general terms to describe the case and do not contain information value for BA. For example, the bug report #92 shown in Figure 1 is reported by *ebidel* and is assigned to two different developers –*mikewest* as the indicated assignee, and *paulirish* as the closer, which are both indications of bug-fix[24,25]. The terms *priority-p2*, *original* and *description*, for example, are used by the reporter to give some high-level instructions rather than technical information about the bug. The same author used the same keywords in several other bug reports. Most of these keywords do not contain technical information about the bug and can be rather misleading when calculating the *TF-IDF* score for developers. *Idf* only addresses the case if the terms appear as a boilerplate for all (or many) bug reports. But if this is repeated in a small number of bugs (e.g., this reporter used these keywords as his signature or partial standard, as it

[4]https://github.com/html5rocks/www.html5rocks.com/issues/92
[5]The title of project is "www.html5rocks.com" and its GitHub page is https://github.com/html5rocks/www.html5rocks.com

is here), *idf* just slightly reduces the weight of those terms. Stop-word removal cannot solve the problem. It just removes some of those misleading keywords, not all of them. The same problem exists for other non-technical keywords (e.g., words *issue*, *context* and *update*) that are rather generic terms used for communication purposes. Finally, there might be specific terms which do not appear a lot in the bug reports, but still do not have technical value for the triager. In all the above cases, some domain-specific reasoning is needed to remove non-technical terms.

As a specific solution to target the important keywords, we perform a simple filtering; we use Stack Overflow as a vocabulary of programming terms including more than 46,000 tags. The tags are defined to describe the questions. Each question should have between one to five tags. Tags cover all the important details of software development topics in any programming language. They are high level enough to be used as subjects when searching for questions, and low level enough to contain technical keywords. A *tag* can be as high level as the name of a programming language, framework, library, technology or method, or as low level as the name of an exception (in a specific programming language), error, web service, layout, plugin, widget, or any other programming topic. Stack Overflow tags are very reliable references because they are authentic keywords generated by the community of developers and curated by them over time.

We remove any term that is not a Stack Overflow tag. This way, there is no need to do the stop-words removal, stemming and so on (although this might still be helpful since the root of the terms might end up as Stack Overflow tags). Considering Figure 1 again, the highlighted terms are the Stack Overflow tags. Removing the non Stack Overflow tags not only gets rid of those misleading terms, but also removes the stop-words, etc. and speeds up the matchings.

We consider each developer a document consisting of one or more sub-documents. Each sub-document includes the textual elements of one of the bug reports assigned to that developer in the past. Later, we will use the time of each sub-document as a decay factor for evidence of expertise of that developer regarding the keywords in that sub-document. Each sub-document consists of a bug report's title and description, plus the main languages of the project (after removing non-Stack Overflow tags). We assumed main language of a project as any programming language that contains at least 15% of the lines of code of that project.

We propose our similarity metric that is obtained from *TF-IDF*, but is time-aware and relies on fixed term weights, to give a score to each developer.

3.1 | *TF-IDF*

The traditional *TF-IDF* [14,43] is an IR measure of the similarity between a query and a document. Having several documents, the document with highest similarity score with the given query is considered the document most similar (and most relevant as a response) to the given query. In the generic form of *TF-IDF* [14], having a query *q*, the score for document d in the corpus D is as follows:

$$\text{score}(q,\ d)\ =\ \sum_{t \in q} \text{idf}(t, D) \cdot \text{tf}(t,\ d) \tag{1}$$

Note that this is assuming that each term appears only once in *q*. To generalize this to include repeats of the terms, the measure becomes as follows:

$$\text{score}(q,\ d)\ =\ \sum_{i=1}^{n} \text{freq}(t_i,\ q) \cdot \text{idf}(t_i, D) \cdot \text{tf}(t_i,\ d) \tag{2}$$

$$n : \text{number of distinct terms in } q$$

In the above formula, $\text{freq}(t_i,\ q)$ is the frequency of the term t_i in query q. The promises of *TF-IDF* is to differentiate between general and specific terms by calculating two main components *tf* and *idf*; *tf* (term frequency) measures number of times a term, t_i, appears in document d, normalized by document length:

$$tf(t_i,\ d)\ =\ \frac{\#\ of\ times\ t_i\ is\ mentioned\ in\ d}{total\ \#\ of\ terms\ in\ d} \tag{3}$$

On the other hand, *idf* (inverse document frequency) measures the importance of a term with regard to all documents in the corpus D:

$$idf(t_i, D) = \log_{10}\left(\frac{total\ \#\ of\ documents\ in\ the\ corpus}{\#\ of\ documents\ containing\ t_i}\right) \tag{4}$$

Regarding occurrence of a term "more than once" in the query or a document, there are alternatives for the relative factors –i.e., $\text{freq}(t_i,\ q)$ and $\text{tf}(t_i,\ d)$. For this purpose, different weighting schemes are used to normalize these two factors [14,44]. For example, occurrence of a term twice may or may not double the value related to that term. We will embed a few of these normalization factors later in our enhanced version for tuning.

With the original *TF-IDF* metric, the documents in a corpus can be ranked based on their similarity with a given query. However, since a document (developer) contains several sub-documents (e.g., the text of previously fixed bug reports) related to different times, we adapt the formula in two steps to be applicable for our problem as mentioned in the following sub-sections.

3.2 | Focusing on a Vocabulary of Terms

We make two changes to the *TF-IDF* (one for each part; *idf* and *tf*), to make it more suitable for our problem.

First, while the current *idf* formula emphasizes specific keywords rather than general ones, it still does not capture *technical* terms, which are important *software programming* keywords. Moreover, it may not give the best weighting for the early stages of the project, when there are not enough bug reports. So, a bigger vocabulary (than the increasing set of documents as the *corpus*) is needed to decide about technical value, specificity or generality of a term. As a result, we use an alternative weighting instead of inferring *idf* from the corpus D; we replace $\text{idf}(t_i, D)$ in the Equation 2 by $w(t_i)$, and calculate the score of document (developer) d for query (bug) q, containing n distinct terms (t_i; *i=1* to *n*) as follows:

$$\text{score}(q,\ d) \ = \sum_{i=1}^{n} \text{freq}(t_i,\ q) \cdot w(t_i) \cdot \text{tf}(t_i,\ d) \tag{5}$$

$$n : \text{number of distinct terms in } q$$

In the above formula, $w(t_i)$ is a generic term weight, independent of any particular project (one time definition) obtained based on frequency of the tag t_i in all Stack Overflow questions. For the term t_i, we define $w(t_i)$ in the similar way $\text{idf}(t_i, D)$ was defined in Equation 4:

$$w(t_i) = log_{10}(\frac{total\ \#\ of\ SO\ questions}{\#\ of\ questions\ tagged\ with\ t_i}) \tag{6}$$

The idea is that the more a tag is repeated in different questions, the less specific the keyword is (because it appears in many situations and becomes a common term). In other words, tags that appear in a small number of questions are specific keywords which can infer a specific problem and are more useful in keyword matching between query q and document d (i.e., the new bug report and a developer).

Stack Overflow covers the used terminology in open-source projects. All the tags are technical, programming keywords curated by expert programmers[6]. So the above formula identifies the technical terms, and, among them, weights the general keywords (that appear frequently in different documents and are not much informative) lower than the specific ones. The keywords that are not a Stack Overflow tag are filtered and removed from the bug reports. In other words, the set of tags of Stack Overflow is considered as the vocabulary of technical terms and the rest of the words are not considered. In addition, our weighting is constant and there is no need to re-calculate it after processing each bug report, unlike the *idf* values (note that the *tf* value still needs to be calculated for each new bug report)[7]. In addition to other technical benefits, the above points lead to speeding up the computations in the assignment process as we will see later.

The second amendment to the original *TF-IDF* is regarding the *tf* formula. Since each document (i.e., developer) consists of several sub-documents (i.e., text of their previously fixed bugs), we calculate the tf value in each of its sub-documents separately, and then aggregate them over all the sub-documents. This, allows considerations (e.g., normalization) based on sub-document length to preserve the effect of all the sub-documents equally. So, the tf will be as follows:

$$\text{tf}(t_i,\ d) \ = \sum_{j=1}^{|SD|} \text{tf}(t_i,\ sd_j) \tag{7}$$

$$SD : \text{number of sub-documents of } d$$

SD is the number of sub-documents of d (i.e., the different assigned bugs to a developer in our problem, or generally, any other work evidence of that developer). This will replace the $\text{tf}(t_i, d)$ factor of the scoring function in Equation 5. This granularity in sub-document (previously assigned bug) level rather than document (developer) level also allows considering time of each sub-document in the next step.

3.3 | Recency-aware Term Weighting

Most work in the past considers a developer's expertise on a single element, constructed by aggregating evidence from a variety of tasks that the developer has contributed to the project. In our work, we consider it as a sequence of evidence, called sub-documents.

The sub-documents (which are actually the textual elements of previous bugs assigned to a developer) belong to different times (from years ago to a previous moment). Over time, developers' interest, their expertise and even their assigned module change. In fact, there is decay in expertise and change in interest areas of developers[45,17,15,46]. Evidence of expertise for developer *d1* about subject matter *x* that occurred last week is more important than other evidence for developer *d2* about *x* that occurred last year.

In order to capture the time of usage of the keywords in calculating the *tf* in Equation 7, which affects the total score of Equation 5, we inject a *recency* factor. The idea is to weigh the recent sub-documents of document (developer) d higher than the old ones. Intuitively, expertise of people

[6] Only top Stack Overflow users with a minimum reputation of 1,500 can create new tags or manage them

[7] Note that for higher performance, the term weights, $w(t_i)$, can be re-calculated once every while (e.g., every few months), to consider the new Stack Overflow tags that might emerge.

decays over time exponentially[47]. However, according to the literature[48], the decay principle can be implemented in many ways, including time-based and event or activity based. We try two *recency* options for sub-documents based on the previous research. First, we just consider the timestamp of the sub-document, according to Servant and Jones[46]. We calculate the recency of a sub-document as the portion of time up to the time of occurrence of the sub-document to the total time –which is between zero and one. For sd_j, the j^{th} sub-document of d, we define the "absolute time-based" recency (the times are in seconds), $\text{abs\_recency}(sd_j)$ as shown below:

$$\text{abs\_recency}(sd_j) = \frac{\text{date}(sd_j) - \text{beginning date of project}}{\text{today} - \text{beginning date of project}} \tag{8}$$

Alternatively, the recency of a document can be estimated based on the amount of work (e.g., bug fixes) that has been done after a specific evidence of expertise (or interest), as a decay factor of that expertise[45]. We calculate the recency as the inverse of the number of sub-documents occurred between this sub-document and the bug. The value of this recency factor is again between zero and one. The "relative activity-based" recency factor for sd_j, the j^{th} sub-document of d, is calculated as shown below:

$$\text{rel\_recency}(sd_j) = \frac{1}{1 + \textit{number of other sub-documents occurred after } sd_j} \tag{9}$$

In the above two definitions, since the older documents (i.e., evidence) are out-dated, their weight decreases. On the contrary, the weight for the new documents increases. We will test both these factors in the tuning of our algorithm and compare their efficiency. With the injection of *recency* factor in Equation 7, the $tf(t_i, d)$ for term t_i in document d will be as follows:

$$tf(t_i,\ d) = \sum_{j=1}^{|SD|} tf(t_i,\ sd_j) \cdot \text{recency}(sd_j)$$
$$SD:\ \text{number of sub-documents of d} \tag{10}$$

Finally, substituting this new definition of $tf(t_i, d)$ in the score function of Equation 5, we obtain $\text{score}(q,\ d)$ as our final similarity metric between query (bug) q and document (developer) d:

$$\text{score}(q,\ d) = \sum_{i=1}^{n} \text{freq}(t_i,\ q) \cdot w(t_i) \cdot \left(\sum_{j=1}^{|SD|} tf(t_i,\ sd_j) \cdot \text{recency}(sd_j) \right)$$
$$n:\ \text{number of distinct terms in q} \tag{11}$$
$$w(t_i):\ \text{term weight}$$
$$SD:\ \text{number of sub-documents in d}$$

We use the above scoring function for assessing the suitability of a developer for a given (new) bug report. We calculate the above score for each member in the developer community. Then sort and rank them based on their obtained score. We call our approach *VTBA* (*Vocabulary and Time based Bug-Assignment*). We cross-validate *VTBA* against real BA data from several open source projects regarding different evaluation measures as described in the next sections.

4 | THE DATA SET

In our previous research, we studied 20 projects, chosen due to their popularity (i.e., number of community members, forks and so on) in GitHub[45] as well as availability of their issues publicly. In this paper, we used the data for the same projects. One limitation of that data set was that it was old and did not include the recent bug reports. We updated the data set so that it is inclusive of all the bugs in the old data set as well as more recent bug reports. Another limitation that we resolved is that in our previous data set[45], we limited the developers to only the shared ones between Stack Overflow and GitHub, but here, *VTBA* does not have such a dependency on Stack Overflow and we do not filter the data set at all. There were 20 projects in our previous study, out of which we could download the information of bug reports of 13 projects[8]. The other 7 projects did not publish their issues anymore (i.e., switched to private issue tracking systems or just revoked public access to their issues in GitHub). To obtain the data, we used GitHub APIs and wrote a set of Javascript programs –to extract the data of those projects– which we made available online[9].

[8]The link to the 13 projects we studied are:
http://github.com/lift/framework, http://github.com/html5rocks/www.html5rocks.com, http://github.com/yui/yui3, http://github.com/khan/khan-exercises, http://github.com/tryghost/ghost, http://github.com/fog/fog, http://github.com/julialang/julia, http://github.com/adobe/brackets, http://github.com/travis-ci/travis-ci, http://github.com/elastic/elasticsearch, http://github.com/saltstack/salt, http://github.com/angular/angular.js and http://github.com/rails/rails

[9]https://github.com/TaskAssignment/software-expertise

We extracted data of those 13 projects from their beginning (2009-04-28 or later) to 2016-10-31. Table 2 shows statistics of our data set. In terms of number of bug reports and especially number of developers in the projects, it is one of the most extensive data sets, comparing with the respective values in other research (as we will see later in Table 5).

TABLE 2 Data set including 13 projects in GitHub (sorted by number of bug-assignments)

Project	Number of members in Developer Community (DC)	Number of members in DC who have ever fixed any bugs	# of bugs	# of bug-assignments
Framework	75	38	325	566
Khan-exercises	206	82	624	857
Yui3	175	77	526	902
Html5rocks	159	47	627	998
Fog	770	208	1,124	1,327
Ghost	473	357	3,578	6,142
Travis-ci	1,159	1,096	5,473	6,334
Angular.js	2,386	1,069	7,402	9,658
Brackets	864	646	6,255	10,462
Rails	4,079	1,431	8,794	11,305
Julia	831	541	9,086	12,748
Salt	2,283	1,227	10,237	15,533
Elasticsearch	1,262	758	10,423	16,184
Total	Average: 1,132 Median: 831	Average: 583 Median: 541	64,474	93,016

According to our previous survey on bug-assignment [24,25], the complete notion of bug-fixer (real assignee) includes anyone who worked towards fixing the bug in any of the four manners; 1) the author of a commit referencing a bug as resolved, 2) the committer[10] (co-author) of a commit referencing a bug as resolved, 3) the developer who closes the bug and 4) the developer who is tagged as "assignee" when the bug is closed. So practically, a bug report can have several real-assignees at different points in time (e.g., a bug report is closed and opened again several times, each time assigned to a developer who does some work), all of which are proper fixers to be considered as ground truth [34,24].

We adopted the above comprehensive definition of real assignee and followed the same instructions as [24,25], including investigation of commit messages and history of the bug reports. For example, for those bug-fix work that are related to the commits, we needed to check the commit messages. According to GitHub documentaion [49], the fixers reference the bugs from commit messages with one of the following specific keywords (or the capital cases of any of the keywords), followed by an optional space, followed by number sign (#) and one or more bug number(s)[11]:

- "fix", "fixes", "fixing", "fixed"
- "close", "closes", "closing", "closed"
- "resolve", "resolves", "resolving", "resolved"

After extracting the list of assignees from commit messages, in order to avoid capturing of communications between developers or typos as bug resolving, we double-checked them against the issue events[12] (which were also extracted by our Java code and GitHub APIs). Similarly, we checked issue events for the other types of evidence –closed, assigned, etc. In total, we found that each bug report in our data set were assigned between 1 and 11 times (average = 1.44) to different developers as *assignee*.

Regarding the community of developers that could be considered as assignees, we extracted and stored information of all the project members in GitHub[13]. In each project, we run the algorithm for all its members, rank them for each bug report and recommend the top ones.

[10] Note that in most cases the author and committer of a commit are the same, but they can be different; the author of a commit, is the developer who actually writes the code. The committer, is the developer who commits this code (the developer who does the merge, rebase, review or approval).

[11] Bug numbers are iterative numbers, usually starting from 1. Pull requests also share this numbering system with bug reports in a manner that a number is considered only for a pull request or a bug report (called issue in GitHub).

[12] Issue events include every interaction about bugs including opening, closing, referencing, subscribing, assigning and reopening.

[13] GitHub has APIs for returning members of a project; https://developer.github.com/v3/repos/collaborators/

Regarding the Stack Overflow data, we used the Stack Exchange Data Dump [50] provided by Internet Archive. We only used the tags and posts information, in order to obtain the weights as mentioned before in Equation 6.

Our source code, data set, input and output files (as well as documentation for running the code in simple steps) are available online[14]. It is one of the most comprehensive data sets currently available for bug-assignment regarding selection of projects (i.e., based on popularity in GitHub), duration of projects, number of bug reports and community members.

5 | EVALUATION

To investigate the effectiveness of our approach, we cross-validate the recommendations of our method against the real assignees in the above data set. For each bug in the projects, we use the metric defined in Equation 11 to assign a relevance score to each developer member of the potential-assignees community, and then sort them based on this score from high to low to recommend a ranked list of developers. Once a bug is processed, we update the list of real assignees associated with this bug in all future assignments, to include all past assignees. Based on the position of the real assignee(s) in our recommended ranked list, we evaluate and report the effectiveness of our approach.

Note that there is no need to divide the data set to training and test sets. That would increase our accuracy (since it is more difficult to predict the real assignees for the first bug reports with no or little previous evidence), but limit its usage. In fact, our approach is on-line; the first few assignments would be less accurate, but while it proceeds with any new assignment, it updates the documents (technical terms in bug reports fixed by each developer) as the expertise profiles of developers.

We perform two types of evaluation; first, we compare *VTBA* against three other methods we implemented; *TF-IDF*, *Time-TF-IDF* (Section 6.1) and *TBA* –Time based Bug-Assignment– which is a simple version of our approach that only considers all the occurrences of the keywords in the past without Stack Overflow based term-weighting. Then, we compare our results against the reported results of 13 other studies in the literature (Section 6.2). Before doing so, in the rest of this section, we introduce the metrics we used for evaluation, and evaluate the impact of parameters in the formulation of *VTBA*.

5.1 | Evaluation Metrics

According to our previous survey on BA the best evaluation measure for BA is Mean Average Precision (MAP) [24,25]. It has four important characteristics: 1) it is a single-figure measure and can be interpreted independently (unlike precision and recall that need to be interpreted together), 2) focuses on high ranks rather than low ranks (it is sensitive to high ranks and penalizes the mistakes in the first ranks more than the next ranks in the recommended list of developers), 3) considers ranks of all the real assignees of each bug and 4) the number of real assignees does not bias in its results since it considers all the real assignees, not only one of them. We mostly use MAP for our analyses and comparisons. After MAP, there are 10 other widely used metrics in the literature (i.e., MAP, top-1, top-5 and top-10 accuracy, precision @1, precision @5 and precision @10, recall @1, recall @5 and recall @10 and MRR) that are recommended for reporting the results:

- Mean Average Precision (MAP): First, calculates the average precision over the ranks of all the real assignees of each bug. Then calculates the mean value of these *average precisions* over all the bug reports.
- top-k accuracy (k = 1, 5 and 10): For each bug report, examining the list of recommended developers by the BA method, if the real assignee is among the top k recommended ones, then it is considered a hit. The hit percentage over all the bug reports is called top-k accuracy. Researchers usually report top-1, top-5 and top-10 accuracies, so we report top-k at these three levels. A few other studies reported top-3, top-20, etc. (not considered in our paper).
- Precision @k (k = 1, 5 and 10): The (average) fraction of real assignees recommended in the top k ranks to the number of recommended developers (k). Usually precision is reported at values 1, 5 and 10 (but there are also less-frequent ones like 3, 20 and 22).
- Recall @k (k = 1, 5 and 10): The (average) fraction of real assignees recommended in the top k recommended developers to the total number of real assignees. Like precision, the most common values for k are 1, 5 and 10 (although other values like 3 and 20 are also reported).
- Mean Reciprocal Rank (MRR): This is the average of the reciprocal rank of the real assignee in recommendation of our approach over all the bug reports. If the bug report has more than one real-assignee, then the highest rank is considered, and the others are ignored.

[14]The data set as well as source code, documentations and detailed output results are available at: **https://github.com/TaskAssignment/TTBA-Outline**

In order to enable the reproducibility of our work, we report our results based on all these metrics (11 metrics in total) for our experiments including a comprehensive set of projects and bug reports. In addition to implementing two other methods and comparing our results against them on the same data, we report a meta-analysis by comparing our results against previous studies, using their reported results on any of these 11 metrics.

5.2 | Parameter Tuning

Tuning is a practice that has been applied to many software engineering studies to configure their parameters[51]. Regarding *VTBA* there are a number of factors about how to run the algorithm that can affect the outcome of our approach. We considered a number of these factors in a tuning experiment on three test projects[15] to obtain the best configuration for our evaluation metric. We mention the main ones called *primary factors* here (the full list of considered factors as well as the detailed process of tuning is available in Appendix A). To tune our approach regarding these factors, we selected a few possible values for each of those factors –based on literature and also our previous experience in optimizing assignment problems[14,41,46]:

- **Weighting**: The term-weighting provided by **w(ti)** provides an option to emphasize on specific keywords in our main scoring function. This factor has two options; "do not use term-weighting" and "use term-weighting". The former includes the constant value of "one" for all the terms. In the later case, we consider the term weighting obtained from Stack Overflow as a vocabulary to emphasize on specific keywords.
- **Recency**: The recency factor provided by $\mathbf{recency(sd_j)}$ enables emphasis on recent usage of keywords in our main scoring function. It has three options; "no recency", "abs_recency" and "rel_recency". The first option disregards the time of usage of the keywords and applies a constant value of "1" for the $recency(sd_j)$ factor in our scoring function. The next two options are as described before in Equations 8 and 9.

We used three projects (one from each category; small, medium and large) for tuning the above parameters, using extensive 2d exploration[52,53]. We performed a mutation experiment (an experiment in which different mutations of various parameters are put together to run and obtain the results in all different possible configurations) to obtain the best configuration for the above parameters. We considered all the possibilities of the parameters and obtained the results for the three test projects on all those combinations. The tuning took two hours in total[16]. The best obtained case is as follows; "use term-weighting" and "rel_recency" (see Table 3 for their tuning results). The best configuration for primary factors is shown in bold and grayed cells in the table. In addition of those factors, there are other less-important factors, which we call *secondary factors*. These factors only have minor effect on MAP and for brevity we omitted them from this table (see Appendix A for details of this mutation experiment as well as the minor effects of secondary factors).

TABLE 3 The results of tuning main factors affecting the accuracy of *VTBA*. The best obtained configuration is highlighted.

Weighting	Recency	MAP
do not use term-weighting	no recency	36.85
	abs_recency	38.65
	rel_recency	54.05
use term-weighting	no recency	37.28
	abs_recency	39.33
	rel_recency	**54.47**

The *recency* makes the most difference in the results. Especially rel_recency which is based on relative amount of work is more effective than abs_recency, which is solely based on time. Using either of rel_recency or abs_recency is better than having no recency. The other effective parameter is *term-weighting*. Having term-weighting increases the MAP in all combinations of the other parameter.

We just used the three test projects in the above steps and did not tune our method on other projects since it is a time-consuming task. Also, it would be unfair to tune on the whole data set. However, doing so, even better overall results might be obtained. In the next section, we use the

[15]We divided the projects into three categories; projects with 1k-, 1k to 10k and 10k+ bug-assignments. There are 4, 4 and 5 projects respectively in small, medium and large categories. Then, we selected the smallest of each category for tuning. The idea was to tune on relatively low number of bugs and leave more bug reports for the main experiment. The selected projects are: lift/framework, fog/fog and adobe/brackets. The percentage of bug reports used for tuning is ∼13%.

[16]This time includes tuning for primary factors (as shown here) and secondary factors (as shown in the Appendix). The primary factors are those whose manipulation has higher effect on the results, comparing to the secondary factors. All the experiments and tunings are done on a machine with Core i7 CPU and 16GB of RAM.

obtained values from the tuning step as the final experiment configuration. In order to avoid bias, we exclude the three test projects, and run the main experiment on the remaining 10 projects.

6 | RESULTS

In this section, we compare the performance of our method (a) against two baseline methods on the same data set, and (b) against the performance of other methods as reported in the literature, as a quick meta-analysis.

6.1 | Comparisons Against Implemented Baseline Methods

We implemented the baseline *TF-IDF* method as well as *Time-TF-IDF* method [15] and compare our results against them. Also, we implemented *TBA* – Time based Bug-Assignment– which is a simple version of our approach that only considers all the occurrences of the keywords in the past without Stack Overflow based term-weighting. The idea is to check the effect of each of the two contributions of the paper (considering the time of all the occurrences of each keyword instead of last occurrence, and, Stack Overflow based term weighting). The final project-based results as well as overall results[17] are shown in Table 4. To avoid bias, we excluded the three test projects in this step. The overall MAP over the other 10 projects is even (around 2%[18]) better than the MAP over three test projects –our tuning. Our MAP ranges between 50% and 71% in different projects with an overall value of 56%. Full details of all 93k bug-assignments as well as Java source code of the implementation of the three methods are available in our repository[14].

Considering overall results (over 10 projects) regarding all the 11 metrics, our approach, *VTBA*, outperforms the three other implemented methods.

VTBA thoroughly outperforms the baseline *TF-IDF*; regarding all the metrics (i.e., *MAP*, *top-k accuracy*, *precision @k*, *recall @k* and MRR), in all single projects as well as overall values, *VTBA* obtained better results. The per-project MAP values for the baseline *TF-IDF* method are between 32% and 59% (which are between 5% to 25% lower than the MAP values of *VTBA* in the same projects) and the overall MAP is 40% (16% lower than *VTBA*).

Also, *VTBA* considerably outperforms *Time-TF-IDF*, regarding most metrics, and most projects. The project-specific MAP values in *Time-TF-IDF* range from 41% to 66% (which in 9 projects are between 3% to 11% lower than MAP values of *VTBA* and in one small project 5% higher) and the overall MAP is 48% (8% lower than *VTBA*).

Finally, comparing *VTBA* against *TBA*, *VTBA* slightly works better than *TBA*. Regarding project-specific results, again, in most projects *VTBA* outperforms *TBA* but in some cases *TBA* competes very lightly.

Generally, *VTBA* works much better in bigger projects. Excluding Yui3 and a small number of project-specific results, *VTBA* works better regarding all the other metrics. We will discuss more about these results in the Discussions (Section 7).

6.2 | Comparison Against Results Reported in the Literature

Different bug-assignment studies reported the performance of their methods on different metrics. In addition to this difference, the assumptions and settings of various studies differ a lot (e.g., definition of real assignee and developer community and filtering inactive developers). These variation points make it hard to compare a new method against previous approaches or aggregate their results [54]. For example, the used data sets might influence the results more than the proposed technical solution. However, despite these difficulties, it is still beneficial to drive a high-level perspective about the previously reported results by the researchers. For this reason, we studied a set of previous research papers and compared their reported results against the results of our method. In our previous research [24,25], we reported 13 exemplary BA studies (the 13 state-of-the-art approaches, mentioned in Table 1) as good candidates for further comparison and meta-analysis. They have quite good number of bugs and developers in their projects and did not have major data filtering. Here, we compare our results against their published results.

Since we obtained our results based on various metrics in addition to MAP (i.e., *top-k accuracy*, *precision @k*, *recall @k* and MRR), we are able to compare the performance of our method against all those studies (regarding one or more metrics for each study). Each of these studies reported their results on their chosen projects, without providing average results over all bug reports in all their projects. Thus, it is hard to make a pairwise comparison between any two studies. So, we compare our final, overall values (which is obtained over all bug reports of all our projects) against results of each project in those baseline approaches. Table 5 shows this comparison.

[17]The "Total" values shows the overall results of all the assignments, assuming all of them as a single, big project and aggregates them altogether. The "Max" values (last row) show the maximum values over the 10 projects.

[18]Note that in all the comparisons of this paper, the absolute difference is mentioned, not relative difference.

TABLE 4 Comparison of results of our approach, VTBA, against three other implemented methods, *TF-IDF*, *Time-TF-IDF*[15] and *TBA* (the simple version of our approach), over 10 projects (sorted by size, from small to large)

Project	Method	Top 1	Top 5	Top 10	p@1 / r@1	p@5 / r@5	p@10 / r@10	MRR	MAP
Khan-exercises	TF-IDF	25.79	68.26	85.06	25.79 / 22.35	15.61 / 66.28	9.82 / 84.34	0.43	41.79
	Time-TF-IDF	39.56	81.68	87.98	39.56 / 34.89	18.69 / 80.32	10.16 / 87.50	0.57	55.45
	TBA	55.54	85.30	**89.38**	55.54 / 50.19	19.39 / 83.98	**10.28 / 88.78**	0.68	65.90
	VTBA	**57.18**	**85.41**	89.26	**57.18 / 51.65**	**19.51 / 84.29**	**10.28** / 88.72	**0.69**	**66.96**
Yui3	TF-IDF	42.35	73.17	83.15	42.35 / 37.37	16.34 / 69.72	9.61 / 81.32	0.56	53.09
	Time-TF-IDF	**56.76**	**78.71**	**86.25**	**56.76 / 50.02**	**17.87 / 76.03**	**10.02 / 84.89**	**0.66**	**63.62**
	TBA	48.00	75.94	85.14	48.00 / 42.50	16.83 / 72.60	9.76 / 83.38	0.60	57.62
	VTBA	50.11	75.83	84.81	50.11 / 44.33	16.92 / 72.75	9.72 / 82.99	0.62	58.95
Html5 rocks	TF-IDF	46.29	83.07	91.88	46.29 / 39.28	19.76 / 78.84	11.50 / 90.93	0.63	59.86
	Time-TF-IDF	55.21	86.77	**93.09**	55.21 / 47.08	**21.38 / 84.47**	**11.72 / 92.59**	0.69	66.29
	TBA	66.43	**87.27**	92.48	66.43 / **56.63**	20.08 / 81.77	11.43 / 90.87	**0.76**	71.35
	VTBA	**66.63**	87.17	92.69	**66.63** / 56.58	20.32 / 82.24	11.49 / 91.27	**0.76**	**71.45**
Ghost	TF-IDF	55.42	75.27	86.84	55.42 / 42.74	18.38 / 68.23	11.24 / 83.31	0.65	58.33
	Time-TF-IDF	55.45	83.00	90.90	55.45 / 42.75	21.60 / 79.34	12.24 / 90.04	0.68	63.76
	TBA	59.02	85.17	**91.78**	59.02 / 45.50	22.21 / 81.98	**12.27 / 90.70**	**0.70**	66.59
	VTBA	**59.39**	**85.27**	91.66	**59.39 / 45.80**	**22.31 / 82.23**	**12.27** / 90.62	**0.70**	**66.92**
Travis- ci	TF-IDF	41.35	71.20	77.23	41.35 / 39.56	14.91 / 69.83	8.18 / 76.38	0.54	52.63
	Time-TF-IDF	51.83	**75.54**	**79.54**	51.83 / 49.67	**15.94 / 74.48**	**8.45 / 78.84**	0.62	60.74
	TBA	**57.69**	74.80	77.99	**57.69 / 55.30**	15.67 / 73.52	8.25 / 77.09	**0.65**	**63.88**
	VTBA	57.47	74.74	78.09	57.47 / 55.07	15.65 / 73.43	8.25 / 77.17	**0.65**	63.75
Angular.js	TF-IDF	22.05	60.32	81.10	22.05 / 19.70	12.82 / 56.70	9.00 / 79.02	0.39	37.21
	Time-TF-IDF	34.50	73.84	85.72	34.50 / 31.21	15.91 / 70.65	**9.63** / 84.35	0.51	49.20
	TBA	**46.67**	79.83	**86.22**	**46.67 / 42.63**	17.25 / 76.82	9.61 / 84.54	**0.61**	**58.27**
	VTBA	46.47	**79.87**	86.19	46.47 / 42.36	**17.32 / 76.97**	9.62 / **84.57**	0.60	58.13
Rails	TF-IDF	21.80	47.00	62.29	21.80 / 19.70	9.85 / 42.53	6.71 / 57.72	0.35	32.21
	Time-TF-IDF	29.73	62.62	75.04	29.73 / 26.14	13.35 / 57.78	8.39 / 71.49	0.44	41.62
	TBA	41.28	69.70	**78.81**	41.28 / 36.12	15.37 / 65.69	**8.91 / 75.84**	**0.54**	50.71
	VTBA	**41.42**	**69.91**	78.69	**41.42 / 36.13**	**15.43 / 65.89**	8.90 / 75.73	**0.54**	**50.82**
Julia	TF-IDF	33.79	62.18	76.36	33.79 / 30.98	13.35 / 58.72	8.42 / 73.46	0.47	45.32
	Time-TF-IDF	34.21	68.61	80.29	34.21 / 31.34	14.93 / 65.14	9.07 / 78.25	0.49	47.82
	TBA	**45.69**	71.78	80.99	**45.69 / 40.53**	**16.15** / 69.20	9.26 / 79.46	**0.57**	**55.46**
	VTBA	45.67	**71.80**	**81.02**	45.67 / 40.49	**16.15 / 69.24**	**9.27 / 79.51**	**0.57**	55.45
Salt	TF-IDF	20.88	58.91	76.06	20.88 / 16.74	13.17 / 53.50	8.83 / 71.66	0.38	35.22
	Time-TF-IDF	29.26	66.06	79.64	29.26 / 23.89	14.93 / 60.80	9.52 / 76.55	0.46	42.72
	TBA	**43.37**	72.56	81.11	**43.37 / 36.35**	17.01 / 68.52	9.83 / 78.76	**0.56**	**53.03**
	VTBA	43.28	**72.63**	**81.14**	43.28 / 36.23	**17.05 / 68.64**	**9.84 / 78.84**	**0.56**	53.01
Elastic search	TF-IDF	25.28	55.70	72.97	25.28 / 22.92	11.68 / 52.53	7.75 / 70.19	0.40	37.95
	Time-TF-IDF	32.04	67.71	82.19	32.04 / 29.18	14.21 / 64.51	8.80 / 79.68	0.48	46.31
	TBA	47.13	73.13	84.27	47.13 / 43.28	15.46 / 69.99	9.04 / 81.81	**0.59**	56.34
	VTBA	**47.61**	**73.97**	**84.71**	**47.61 / 43.75**	**15.65 / 70.85**	**9.10 / 82.29**	**0.59**	**56.92**
Total (13 proj)	TF-IDF	28.92	60.05	75.44	28.92 / 25.34	13.07 / 56.01	8.46 / 72.32	0.43	40.88
	Time-TF-IDF	35.80	69.84	81.52	35.80 / 31.48	15.42 / 66.14	9.32 / 79.29	0.51	48.55
	TBA	47.37	74.51	82.86	47.37 / 41.77	16.72 / 71.31	9.51 / 80.85	**0.59**	56.68
	VTBA	**47.50**	**74.73**	**82.93**	**47.50 / 41.85**	**16.79 / 71.58**	**9.53 / 80.96**	**0.59**	**56.83**
Max (13 proj)	TF-IDF	55.42	83.07	91.88	55.42 / 42.74	19.76 / 78.84	11.50 / 90.93	0.65	59.86
	Time-TF-IDF	56.76	86.77	**93.09**	56.76 / 50.02	21.60 / **84.47**	12.24 / **92.59**	0.69	66.29
	TBA	66.43	**87.27**	92.48	66.43 / **56.63**	22.21 / 83.98	**12.27** / 90.87	**0.76**	71.35
	VTBA	**66.63**	87.17	92.69	**66.63** / 56.58	**22.31** / 84.29	**12.27** / 91.27	**0.76**	**71.45**

TABLE 5 The evaluation measures and other design choices of the selected research; We mention results up to hundredths which is two decimal digits (except the cases that in the cited research reported less than two decimals). "∼" means the exact value was unclear (e.g., estimated from a figure). "**-**" or *blank cell* means the value is not mentioned nor depicted in the respective research. Similarly, "?" shows a missing but important value (the number was expected but it is not clearly mentioned). The bold values are the highest values in each column.

Method / Project	#devs	#bugs	Top1	Top5	Top10	P@1 / r@1	p@5 / r@5	p@10 / r@10	MRR	MAP
VTBA(our approach)										
Total (13 proj)	75-4,079	566-16,184	47.50	74.73	**82.93**	**47.50/41.85**	**16.79/71.58**	9.53/**80.96**	**0.59**	**56.83**
Max (13 proj)			**66.63**	**87.17**	**92.69**	**66.63/56.58**	**22.31/84.29**	**12.27 / 91.27**	**0.76**	**71.45**
Čubranić and Murphy [28]										
Eclipse	162	15,859	30.00							
Canfora and Cerulo [29]										
Mozilla	637	12,447				- / 12.0	- / 21.0	- / 24.0		
KDE	373	14,396				- / 5.0	- / 10.0	- / 12.0		
Jeong, *et al.* [1]										
Eclipse	1,200	46,426		77.14						
Mozilla	2,400	84,559		70.82						
Matter *et al.* [17]										
Eclipse	210	130,769				33.6 / ∼27	∼16 / ∼59	∼**10** / 71.0		
Tamrawi *et al.* [30,26]										
Firefox	3,014	188,139	32.1	73.9						
Eclipse	2,144	177,637	42.6	**80.1**						
Apache	1,695	43,162	39.8	75.0						
Net Beans	380	23,522	31.8	60.4						
FreeDesktop	374	17,084	**51.2**	81.1						
GCC	293	19,430	48.6	79.2						
Jazz	156	34,220	31.3	75.3						
Bhattacharya and Neamtiu [2], Bhattacharya *et al.* [31,2]										
Mozilla	?	549,962	27.48	77.87						
Eclipse	?	306,297	32.99	77.43						
Shokripour *et al.* [5]										
Eclipse	?	?				- / ∼32	- / ∼71			
Mozilla	?	?				- / ∼27	- / ∼48			
Gnome	?	?				- / ∼10	- / ∼45			
Zhang *et al.* [11]										
Mozilla	∼874	74,100							0.28	44.49
Eclipse	∼544	42,560							0.28	56.42
Ant	∼203	763							0.35	36.48
TomCat6	∼79	489							0.35	36.54
Cavalcanti *et al.* [27], Cavalcanti *et al.* [32]										
New SIAFI - A	70	781	31.40							
New SIAFI - B	70	1031	22.00							
Sun *et al.* [33]										
JEdit	123	?	28.0	60.1	79.8					
Hadoop	82	?	8.5	30.1	50.3					
JDT-Debug	47	?	14.4	46.6	66.4					
Elastic	661	?	13.6	43.6	75.2					
Libgdx	345	?	22.0	51.3	69.6					

On all the metrics, the highest values are related to one of projects in *VTBA*. Regarding the overall (total) values, *VTBA* surpasses the results of all the projects of 8 studies;[28,29,26,5,11,32,27,33]. There are a few cases in other 5 studies that they work partially better than *VTBA*:

Jeong *et al.*[1] obtained around 2% higher top-5 accuracy than our average in Eclipse, but 5% lower in Mozilla. Matter *et al.*[17] has almost the same precision @5 and @10 comparing to *VTBA*, but at least 10% below our average results regarding precision @1, and recall @1, @5 and @10). Tamrawi *et al.*[30] evaluated their results over 7 projects. Their top-1 accuracy in two projects and top-5 accuracy in four projects are slightly better than our average. *VTBA* outperforms their reported results in the rest of the projects. The maximum and average of their values over all projects are lower than our maximum and average. Bhattacharya *et al.*[31,2] obtained slightly (2%) better top-5 accuracy than our average, but our top-1 is around 15% higher than their results in both projects.

These comparisons are interesting and lead us to the conclusion that our approach works well. However, as we mentioned at the beginning of this section, since the data sets and other experiment details differ, we still cannot claim a total superiority of *VTBA* against all of them.

7 | DISCUSSION

We tuned our approach on three projects in our data set and then tested it on the remaining 10 projects. The final results on the 10 projects were even better than the tuning results on 3 tuning projects. This shows that the tuning is general and robust enough.

On average, our method (*VTBA*) has 56% overall MAP. The high obtained MAP in our experiment shows that in most cases the real assignees are successfully retrieved and ranked at the top few ranks of the recommended list. This is also proved by the high values on the other reported metrics. For example, in 47.5% of the cases, *VTBA* recommends one of the real assignees in the first rank, in 74.73% of the cases in the first 5, and, in 82.93% of the cases in the top 10 ranks.

7.1 | Intuitions About Comparison of the Implemented Approaches

Regarding overall results in all the 11 metrics, our approach outperforms the three other implemented methods (*TF-IDF*, *Time-TF-IDF* and *TBA*). Considering project-specific results, *VTBA* outperforms *TF-IDF* and *Time-TF-IDF* in almost all projects. In 7 projects (*Khan-exercises*, *Ghost*, *Julia*, *Elasticsearch*, *Salt* *Angular* and *Rails*), our approach comprehensively works better than the two other methods. In two projects (*Html5rocks* as a small project, and *Travis-ci* as a medium project), regarding some of the metrics, *Time-TF-IDF* works better, but still in MAP and other evaluation measures our approach works better. Note that between available BA evaluation metrics, only MAP is comprehensive enough to include all the assignees of each bug report[24,25]. For example, in *Html5rocks* *Time-TF-IDF* is ∼1% higher than *VTBA* regarding precision @5. However, this metric only considers the first 5 recommendations and ignores the rest, but regarding MAP, *VTBA*'s results are ∼5% higher. So again, in these two projects (*Html5rocks* and *Travis-ci*), our approach outperforms the other two methods. The only exception is *Yui3* in which *Time-TF-IDF* has 5% higher MAP than *VTBA*. The reason can be regarding its size and duration; *Yui3* is the second smallest project, with the smallest duration, i.e., ∼38 months (while the average and median over all projects are 61 and 67 months respectively). So *VTBA*'s high granularity of expertise of developers over time did not work in this project as well as the other projects.

This fine-grained usage of data is especially evident in bigger projects; In four large projects with 10k+ bug-assignments (*Rails*, *Salt*, *Elasticsearch* and *Julia*), our approach obtained ∼10% and ∼20% better results than *Time-TF-IDF* and *TF-IDF* respectively. The reason can be that in big projects, there are more bug-assignment data available, which are scattered over various dates. It enables our fine-grained scoring function to utilize them as various evidence of expertise distributed over time. On the other hand, in small projects, lack of enough bug reports might bring uncertainty in making decisions. *Yui3*, as the only project in which *Time-TF-IDF* works better than our approach, is a small project. *Html5rocks* and *Travis-ci*, in which in some measures (excluding MAP) *Time-TF-IDF* competes against our approach are respectively small and medium projects.

VTBA slightly works better than the other version of our approach, *TBA*. This shows that the fine-grained view over the usage of the keywords over time (all usages of a keyword by the developers rather than the last one) accounts for most of the improvements, although the Stack Overflow based term weighting is also important.

We captured the details of the system we ran our experiments on. On a computer with Core i7 CPU and 16GB of RAM, it took between 1 to 26 seconds to run for each project (average = 12s), depending on the number of bugs to be assigned. We found that it is quite fast; the average time for assigning developers to each bug, considering the data of all projects, is 1.7 milliseconds. In total, it took 138 seconds for doing all the 93k bug-assignments. The two other approaches, *TF-IDF* and *Time-TF-IDF*, took 413 and 465 seconds respectively. There are two reasons for this speed up in our approach against *TF-IDF* and *Time-TF-IDF*. First, our approach does not re-calculate the *idf* values after processing each bug report (which includes updating indexes and statistics related to frequency of the terms in the corpus); instead, it replaces it by $w(t_i)$ which is calculated once over the Stack Overflow data as pre-processing (although this does not need to be calculated every time, we measured its time; the pre-processing on Stack Overflow data took 63 seconds, which is again quite fast). Note that our approach still needs to calculate the *tf* values for all

sub-documents (i.e., previously assigned bug reports to a developer). Second, *VTBA* filters the non-technical terms (i.e., any term except the Stack Overflow tags), and in fact, shortens the bug reports. As a result of these two enhancements, *VTBA* works more than three times faster than the other two compared methods.

7.2 | Intuitions About Comparison of Reported Results

From a higher perspective, we compared the *VTBA* results against the reported meta-data of 13 best studies in the field. We showed that according to the reported results, *VTBA* outperforms 8 studies. Five studies obtained higher results regarding some metrics in a few projects, but we achieved much better results regarding several other metrics / projects. Note that over all the metrics, the highest values are obtained in one of our projects. However, we admit that this superiority is not total and project or experiment limitations (like data set characteristics) might have affected this outcome. Below, we compare some of these dimensions.

Regarding the dimensions of variability of the data sets (number of projects, bug reports and developers), our data set includes more data. It includes abundant bug reports, more than most previous research in the area. With respect to number of developers per project (which is a key factor in the difficulty of assignee recommendation in a project), our data set includes the most populated projects (on average, 1,132 developers per project). So, the overall situations of our data set is more difficult than most mentioned studies in Table 5. Also the data we use is the simplest form of data used for BA (only bug reports' title and descriptions). As a result, competing against the meta-data reported by them as mentioned in the above table seems fair and reasonable.

Regarding limitations, unlike some other studies mentioned in Table 5 [17,5,33], there is no dependency on specific information (e.g., interaction histories between developers, heavy historical changes on the source code files, the component and severity of the bug, etc.), other than title and description of the bug reports, which are easily accessible in any open-source repository.

Unlike most previous research that we reported their results in Table 5 [28,1,17,30,26,31,2,32,27], our approach does not need big training data sets. Most approaches utilize ML algorithms and require big training steps, which make their approach incapable of recommending assignees –with high accuracy– for the first portion of their data set. We tested our approach from the first bug, to the last one, with no training data. In fact, the training data of our approach includes every bug report is being processed for assignment in our main experiment. For predicting the assignee for bug report #n, the information of n-1 previous bug reports were considered. Starting from the first bug report, every bug it processes, it builds the sub-documents (i.e., the text of the bug reports fixed by each developer up to that point in time) as the expertise profiles of developers and uses them in assigning developers for next bugs. Considering the first part of the data set as training set (like those studies) would even increase our accuracy.

Considering the fact that the above research were the most reproducible research selected for comparison and meta-analysis [24], and aggregating with the obtained results regarding superiority of our approach against two other implemented methods (*TF-IDF* and *Time-TF-IDF*), we argue that *VTBA* is capable of precise assignee recommendation.

7.3 | Implications for practice

Considering the practicality of our method, there are some points that need consideration before applying the method in practice. Some of these implications are listed here.

The companies and projects have different issue-handling approaches and settings. When applying *VTBA* on these projects, the outcome might be affected by those approaches and settings. For example, some of them have a "push" approach in which project administrators assign bugs and other similar tasks to the developers to fix or implement. Others adopt a "pull" method in which the developers pull or pick an item from a queue of tasks/bugs. Most Scrum open-source projects favor for pull-based approaches [55,56]. Similarly, our approach works better for those projects since less restrictions are applied from outside project. In other words, the most important factor in selecting a developer to fix a bug is the developers' expertise and interest. In these cases, the best results of *VTBA* is expected. However, when the level of automation increases and more factors are being considered, we cannot expect the same performance (like in push-based approaches, in which the administrators' orders and decisions are important as well as the previous background of the developers). As the level of automation increases, more factors are to be considered like developers' workload and wage. These are the factors that are not currently covered by *VTBA*. In fact, to consider those factors, the project management team needs to use the recommendations of *VTBA* along with other restrictions in a semi-supervised manner (e.g., for each task or bug, pick a few developers from the top of the recommended list and apply those restrictions manually to select the best person for handling the task). Those restrictions and decisions can be embedded in the *VTBA* formula and model in a separate study, to cover multiple objectives in BA, as did some researchers before [23,21].

7.4 | Threats to Validity

Like any other research, there are some possible challenges and threats to validity that we address below.

7.4.1 | Internal validity

1) In order to capture the real assignees (i.e., the golden truth to validate our approach), we used projects that use GitHub's issue tracker. We looked for bugs in GitHub projects and their certain links to the commits and issue (bug) events, to find the real assignees. Although it is a common practice to mention and preserve those links in GitHub's open-source projects [49], there might be some missing links [57] that are not considered here.

We admit that ignoring the missing links can lightly affect the validation of our approach. Note that validation of these cases and inclusion of the full links between commits and bug reports is a tedious task, that needs to be done manually [57]. To the best of our knowledge, no previous research has done this manual process. All the previous research in the field cross-validated their approach against heuristic-based golden-truth extracted automatically from available data of software projects.

2) We found that there are some bug reports that have no, or only a few, Stack Overflow tags. If there are many such bugs in a project, the effectiveness of our approach will suffer.

To address this issue, we programmatically counted the tags in each bug report. The bug reports of each project have 20 tags on average (between 11 to 32 tags in average in each project, with median 19; see Figure 1 for an example). This number of tags is more than enough to convey the idea of a bug report. Only around 1% of the bugs have fewer than three tags and 5% have fewer than five tags. 75% of the bug reports have ten or more tags. For extreme cases where not enough tags are included in the text of the bug reports, we outline some potential means to expand the keywords in Section 8 as a future work.

3) There may be very short tags that are usually used along with other tags (e.g., "r" that usually is used with "rstudio", "plot", "sentiment-analysis", etc). In Stack Overflow, they are also explained by the question description. Their use may be problematic (e.g., the user may mean the name of a simple variable, not the meaning indicated by the tag). In addition, some tags are generic words that have specific meanings (like the term "this", which is also a *Java* keyword) and may be misrepresenting.

To address this issue, first, we checked the length of all the tags; there are in total 46,278 tags. Among those, 11 and 152 tags are one- and two-letter tags that usually come with other tags and it is hard to use them as indication of meaning outside of Stack Overflow. To avoid false positives, these tags can simply be ignored since their number is low. The rest of them, 46,115 tags in total, are considered to filter the bug reports' textual elements (1,730 of them have three letters and the rest have four or more letters). Note that, as mentioned in the previous issue, usually (in 99% of the cases) there are enough tags in the text of the bug report. Moreover, in the case of misleading tags, again, we refer to the above statistics; since there are enough tags in each bug report, the combination of remaining terms usually is enough to converge toward the general idea of the bug report and usually cover the possible false positives. So, some of them may be eliminated through a stop-word removal step, as we mention in the future works.

4) Some tags have synonyms in Stack Overflow which are identified by the expert users. We did not merge those synonyms. For example, "crypto" is a synonym for "cryptography". Merging the two includes changing all occurrences of "crypto" in the textual elements of all the bug reports to "cryptography". Also, it needs to merge the statistics of usages of the two terms into one term in Stack Overflow. In addition, synonym suggestion techniques [58] can be used for even enhancing the accuracy. These updates might change the weights and may lead to better accuracy but needs extra effort.

While we agree that this can be considered as future work, we anticipate minor effect on the outcome since the number of synonym pairs in Stack Overflow is quite low, compared to the total number of tags [58].

5) Some of the tags are combinations of several keywords –e.g., "pull-request", "active-record-query", "for-loop", "jdk1.8.0", "http-status-code-403" and "elasticsearch-java-api", but they are not consistently used by everyone. The developers may combine these keywords the same way Stack Overflow does, or in a different manner (e.g., with capitalization, space, underscore and so on). Consider the tag "jdk1.8.0" as an example; it may be found as "jdk 1.8.0", "jdk_1.8.0", or "jdk 1 8 0".

To address this issue, we did a cleaning step before running our approach; considering the textual elements of the bug reports, we applied a set of heuristics that concatenate consequent keywords with/without valid connections (i.e., "." and "-") to build possible composite tags. To avoid false positives, we have put limitations for the length of the combined terms. In the end, we have made joint tags from two, three and four consecutive keywords in the bug reports (respectively 11,999, 1,008 and 51 Stack Overflow tags with 236,905, 5,746 and 228 total occurrences in the bug reports).

7.4.2 | External validity

1) One aspect that can diminish the performance of our approach is the appearance of new Stack Overflow tags. Since our term weights are calculated only once, they do not update during time.

We answer that this can only have a negative effect on the accuracy of our approach in the long term. It takes a long time (e.g., several months) to appear a set of new programming concepts and keywords, being adopted by Stack Overflow users as tags, and being used by the developers in the description of bug reports. Moreover, to address this issue in the long term, we can re-calculate the term weights once every few months, based on a new set of tags in Stack Overflow.

2) One may mention that VTBA does not assign any bugs to the newcomers since they have no previously assigned bugs, hence they have no sub-documents. So, they are given a score of zero.

We argue that this is true for any bug-assignment method, because at each point in time, they only assign bugs to the developers who have some evidence of expertise in the system, up to that point in time. When predicting the assignee for a bug that is assigned to a newcomer, *d1*, like any other bug-assignment method, *VTBA* fails to predict the correct assignee since there is no previous evidence for *d1*. It gives the score of zero to *d1*. However, after this point in time, there will be one piece of evidence for *d1*, and *VTBA* calculates a positive score for *d1*. With processing of more bugs having *d1* as the real assignee, more sub-documents are stored in the data set as his evidence of expertise, and his chance of being recommended increases.

3) While we tested our approach using several open-source projects and their available data, one could suspect its usefulness in other projects like proprietary projects within companies. For example, the term-weighting of our scoring function (Equation 6) might be questionable in proprietary software.

We admit that our final scoring function (Equation 11) might not work perfectly in new settings and projects (like proprietary software). However, the method has the capability of being tuned for different situations, which can be researched in a separate study. For example, since for those specific projects, Stack Overflow tags may not be representative of the most important keywords, other *idf*-like weighting schemes with proper indexing can be utilized from their developer network. Even the other factors (i.e., $\mathrm{freq}(t_i, q)$ and $\mathrm{tf}(t_i, d)$) can be utilized in a different way that works better for those projects. All these adjustments can be met with a small percentage of bug reports in the project, or a few test projects, to obtain desirable results in different setups. The important aspect is that the general scheme of our approach supports granularity of the technical terms and their time of usage by the developers as well as some term-weights obtained from a vocabulary of terms. Utilization of such vocabulary from a developer network to extract a set of mostly referred keywords can help to obtain the required domain-specific term weights.

4) Finally, there are specific cases in which the *VTBA* does not work as described above (at least in special situations). For example, if a single developer does everything related to some Stack Overflow tags, *VTBA* recommends that developer for all the related bugs (even though they might be too many bugs). This might imbalance the workload of the developers. More importantly, if that developer leaves the project, there wouldn't be good recommendations by *VTBA* and the team will be in trouble in fixing related bugs (i.e., the "Truck Factor" [59], or, the number of developers whose removal from the project makes trouble, is too low).

We admit that *VTBA* does not consider such situations and just recommends developers based on their expertise and interest, which are obtained from previous bug-fixing history. To address this issue, the project management team needs to make some knowledge distribution policies among the team members. This way, they will not be dependent on a specific developer or a few number of them (i.e., the truck factor will be increased). In any case, the simple scoring function of *VTBA* allows the project managers to embed it in their multi-objective function (or use it somehow) to have a higher level of automation for the project (e.g., balance the workload or distribute the knowledge between developers).

8 | CONCLUSIONS AND FUTURE WORK

In this paper, we described *VTBA*, a new BA approach based on developers' previous contributions in project. We used Stack Overflow as a vocabulary of technical terms to identify the importance and specificity of keywords and utilized it along with recency of developers' work in a scoring function for BA. We demonstrated that our method outperforms most state-of-the-art methods. *VTBA* does not require training; at each point in time it uses only the previous bug fixes. It might not produce very good results for the first few bug reports of each project but improves quickly by obtaining some BA data and using them as evidence of expertise.

Our BA metric considers previous bug fixing as evidence of expertise of developers, but our preliminary investigations and results (not included in this paper) show that it is expandable to other sources of expertise in the repository like commits. We plan to consider it as a future work.

Our data set is the most comprehensive data set used in this field, to our knowledge. Our data set as well as our code in GitHub are available online[14] for further researchers to compare their method against our approach on our data set. As an extensive data set of bugs and developers, it can also be used for any other BA experiment.

In the future, we plan to expand the tags in a bug report to infer new keywords with query expansion or graph-based methodologies. This way, we may eliminate the limited number of tags in a few percentage of the bug reports and enhance the method's accuracy. Finally, some additional stop-words removal (e.g., removing *generic* Stack Overflow tags like *this*, *for* or *while*) may increase the quality of the tags and enhance the accuracy of our approach.

Currently, for each tag, we considered weights based on their appearance in Stack Overflow. Then, we used these weights constantly for all the projects. This can be changed to obtain a better accuracy. One would envision using more specific weighted keywords for each domain or project. This can be done by obtaining the weights directly using the information within domain or project (instead of Stack Overflow). The promises of this paper, however, was to utilize the Stack Overflow as the vocabulary, in addition to a fine-grained utilization of time in the usage of keywords, and we showed that it works for the purpose of BA.

ACKNOWLEDGEMENTS

The work is supported by Graduate Student Scholarship[19] funded by Alberta Innovates - Technology Futures (AITF)[20] and Queen Elizabeth II Graduate Scholarship[21] funded by Faculty of Graduate Studies and Research (FGSR)[22] at University of Alberta.

References

1. Jeong G, Kim S, Zimmermann T. Improving Bug Triage with Bug Tossing Graphs. In: ESEC/FSE '09. ACM; 2009: 111–120.
2. Bhattacharya P, Neamtiu I. Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging. In: IEEE. ; 2010: 1–10.
3. Aljarah I, Banitaan S, Abufardeh S, Jin W, Salem S. Selecting discriminating terms for bug assignment: a formal analysis. In: ACM. ; 2011: 12.
4. Linares-Vásquez M, Hossen K, Dang H, Kagdi H, Gethers M, Poshyvanyk D. Triaging incoming change requests: Bug or commit history, or code authorship?. In: IEEE. ; 2012: 451–460.
5. Shokripour R, Kasirun Z, Zamani S, Anvik J. Automatic Bug Assignment Using Information Extraction Methods. In: ; 2012: 144-149
6. Nguyen TT, Nguyen AT, Nguyen TN. Topic-based Bug Assignment. *SIGSOFT Softw. Eng. Notes* 2014; 39(1): 1–4. doi: 10.1145/2557833.2560585
7. Liu J, Tian Y, Yu X, et al. A Multi-Source Approach for Bug Triage. *International Journal of Software Engineering and Knowledge Engineering* 2016; 26(09n10): 1593–1604.
8. Saha RK, Khurshid S, Perry DE. Understanding the triaging and fixing processes of long lived bugs. *Information and Software Technology* 2015; 65: 114–128.
9. Akbarinasaji S, Caglayan B, Bener A. Predicting bug-fixing time: A replication study using an open source software project. *Journal of Systems and Software* 2017.
10. Tian Y, Wijedasa D, Lo D, Le Gouesy C. Learning to rank for bug report assignee recommendation. In: IEEE. ; 2016: 1–10.
11. Zhang W, Wang S, Wang Q. KSAP: An approach to bug report assignment using KNN search and heterogeneous proximity. *Information and Software Technology* 2016; 70: 68–84.
12. Zhang T, Chen J, Yang G, Lee B, Luo X. Towards more accurate severity prediction and fixer recommendation of software bugs. *Journal of Systems and Software* 2016; 117: 166–184.
13. Robillard M, Maalej W, Walker R, Zimmerman T. *Recommendation systems in software engineering*. Springer . 2014.
14. Manning CD, Raghavan P, Schütze H. *Introduction to Information Retrieval*. Cambridge University Press . 2008.
15. Shokripour R, Anvik J, Kasirun ZM, Zamani S. A time-based approach to automatic bug report assignment. *Journal of Systems and Software* 2015; 102: 109–122.

[19] https://fund.albertainnovates.ca/Fund/BasicResearch/GraduateStudentScholarships.aspx
[20] https://innotechalberta.ca
[21] https://www.ualberta.ca/graduate-studies/awards-and-funding/scholarships/queen-elizabeth-ii
[22] https://www.ualberta.ca/graduate-studies

16. Banerjee S, Syed Z, Helmick J, Culp M, Ryan K, Cukic B. Automated triaging of very large bug repositories. *Information and Software Technology* 2016. doi: https://doi.org/10.1016/j.infsof.2016.09.006

17. Matter D, Kuhn A, Nierstrasz O. Assigning bug reports using a vocabulary-based expertise model of developers. In: IEEE. ; 2009: 131–140.

18. Hu H, Zhang H, Xuan J, Sun W. Effective bug triage based on historical bug-fix information. In: IEEE. ; 2014: 122–132.

19. Jonsson L. Increasing anomaly handling efficiency in large organizations using applied machine learning. In: IEEE. ; 2013: 1361–1364.

20. Jonsson L, Borg M, Broman D, Sandahl K, Eldh S, Runeson P. Automated bug assignment: Ensemble-based machine learning in large scale industrial contexts. *Empirical Software Engineering* 2016; 21(4): 1533–1578.

21. Karim MR, Ruhe G, Rahman M, Garousi V, Zimmermann T, others . An empirical investigation of single-objective and multiobjective evolutionary algorithms for developer's assignment to bugs. *Journal of Software: Evolution and Process* 2016; 28(12): 1025–1060.

22. Rahman MM, Karima MR, Ruhea G, Garousic V, Zimmermannd T. An empirical investigation of a genetic algorithm for developer's assignment to bugs. In: ; 2012: 1–15.

23. Khalil E, Assaf M, Sayyad AS. Human resource optimization for bug fixing: balancing short-term and long-term objectives. In: Springer. ; 2017: 124–129.

24. Sajedi A, Stroulia E. Guidlines for Evaluating Bug-assignment Research. *Accepted for publication (under review), Journal of Software: Evolution and Process* 2019.

25. Sajedi A. *Bug Assignment: Insights on Methods, Data and Evaluation*. PhD thesis. University of Alberta, 2018. pp: 7-52.

26. Tamrawi A, Nguyen TT, Al-Kofahi J, Nguyen TN. Fuzzy set-based automatic bug triaging: NIER track. In: IEEE. ; 2011: 884–887.

27. Cavalcanti YC, Machado IdC, Neto PA, Almeida dES, Meira SRdL. Combining rule-based and information retrieval techniques to assign software change requests. In: ACM. ; 2014: 325–330.

28. Čubranić D, Murphy GC. Automatic bug triage using text categorization. In: Citeseer. KSI Press; 2004: 92–97.

29. Canfora G, Cerulo L. Supporting change request assignment in open source development. In: ACM. ; 2006: 1767–1772.

30. Tamrawi A, Nguyen TT, Al-Kofahi JM, Nguyen TN. Fuzzy set and cache-based approach for bug triaging. In: ACM. ; 2011: 365–375.

31. Bhattacharya P, Neamtiu I, Shelton CR. Automated, highly-accurate, bug assignment using machine learning and tossing graphs. *Journal of Systems and Software* 2012; 85(10): 2275–2292.

32. Cavalcanti YC, Carmo Machado dI, Neto PAdMS, Almeida dES. Towards semi-automated assignment of software change requests. *Journal of Systems and Software* 2016; 115: 82–101.

33. Sun X, Yang H, Xia X, Li B. Enhancing developer recommendation with supplementary information via mining historical commits. *Journal of Systems and Software* 2017; 134: 355–368.

34. Anvik J, Hiew L, Murphy GC. Who should fix this bug?. In: ACM. ; 2006: 361–370.

35. Shokripour R, Anvik J, Kasirun ZM, Zamani S. Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation. In: IEEE Press. ; 2013: 2–11.

36. Sun X, Liu X, Hu J, Zhu J. Empirical studies on the nlp techniques for source code data preprocessing. In: ACM. ; 2014: 32–39.

37. Zanjani MB, Kagdi H, Bird C. Using developer-interaction trails to triage change requests. In: IEEE Press. ; 2015: 88–98.

38. Ponzanelli L, Mocci A, Lanza M. Stormed: Stack overflow ready made data. In: IEEE. ; 2015: 474–477.

39. Meldrum S, Licorish SA, Savarimuthu BTR. Crowdsourced Knowledge on Stack Overflow: A Systematic Mapping Study. In: ACM. ; 2017: 180–185.

40. Li G, Zhu H, Lu T, Ding X, Gu N. Is It Good to Be Like Wikipedia?: Exploring the Trade-offs of Introducing Collaborative Editing Model to Q&A Sites. In: ACM. ; 2015: 1080–1091.

41. Sajedi A, Hindle A, Stroulia E. Crowdsourced Bug Triaging: Leveraging Q&A Platforms for Bug Assignment. In: FASE '16. Springer; 2016: 231–248.

42. Huang L, Deshmukh RA, Mostafa J, Greenberg J. SPIRO-V: A Collaborative Approach to Controlled Vocabularies Gathering and Management. In: JCDL '10. ACM; 2010; New York, NY, USA: 371–372

43. Cavalcanti YC, Mota Silveira Neto PA, Machado IdC, Vale TF, Almeida ES, Meira SRdL. Challenges and opportunities for software change request repositories: a systematic mapping study. *Journal of Software: Evolution and Process* 2014; 26(7): 620–653.

44. Manning CD, Schütze H, others . *Foundations of statistical natural language processing*. 999. MIT Press . 1999.

45. Sajedi A, Hindle A, Stroulia E. Crowdsourced Bug Triaging. In: IEEE; 2015: 506–510.

46. Servant F, Jones JA. WhoseFault: automatic developer-to-fault assignment through fault localization. In: IEEE Press. ; 2012: 36–46.

47. Carlson PE. *Engaging developers in open source software projects: Harnessing social and technical data mining to improve software development*. PhD thesis. Iowa State University, 2015.

48. ElSalamouny E, Krukow KT, Sassone V. An analysis of the exponential decay principle in probabilistic trust models. *Theoretical computer science* 2009; 410(41): 4067–4084.

49. Github . Closing issues using keywords. 2017.

50. Archive I. Stack Exchange Data Dump. 2017.

51. Borg M. TuneR: a framework for tuning software engineering tools with hands-on instructions in R. *Journal of software: Evolution and Process* 2016; 28(6): 427–459.

52. Blanco R, Lioma C. Graph-based term weighting for information retrieval. *Information retrieval* 2012; 15(1): 54–92.

53. Craswell N, Robertson S, Zaragoza H, Taylor M. Relevance weighting for query independent evidence. In: ACM. ; 2005: 416–423.

54. Borg M, Runeson P. IR in software traceability: From a bird's eye view. In: IEEE. ; 2013: 243–246.

55. Overeem B. The 8 Stances of a Scrum Master. tech. rep., Scrum.org; 2017.

56. Havn M. Using Continuous Delivery Maturity Models. tech. rep., AARHUS University; 2018.

57. Bachmann A, Bird C, Rahman F, Devanbu P, Bernstein A. The missing links: bugs and bug-fix commits. In: ACM. ; 2010: 97–106.

58. Beyer S, Pinzger M. Synonym suggestion for tags on stack overflow. In: IEEE Press. ; 2015: 94–103.

59. Berteig M. *Agile Advice - Creating High Performance Teams In Business Organizations*. Berteig Consulting Inc. . 2015.

APPENDIX

A) DETAILS OF MUTATION EXPERIMENT FOR TUNING 6 PARAMETERS OF OUR METHOD

There are a number of factors that can affect the quality of our assignment. We introduce them here, and consider their effect in a small experiment on a few test projects:

- **Weighting**: The term-weighting provided by **w(ti)** provides an option to emphasize on specific keywords in our main scoring function. This factor has two options; "do not use term-weighting" and "use term-weighting". The former includes the constant value of "one" for all the terms. In the later case, we consider the term weighting obtained from Stack Overflow as a vocabulary or thesaurus to emphasize on specific keywords.
- **recency**: The recency factor provided by **recency(sd$_j$)** enables emphasis on recent usage of keywords in our main scoring function. It has three options; "no recency", "abs_recency" and "rel_recency". The first option disregards the time of usage of the keywords and applies a constant value of "1" for the recency(sd$_j$) factor in our scoring function. The next two options are as described before in Equations 8 and 9.
- **tf(t$_i$, sd$_j$)**: having the text of the sub-document sd_j, term-frequency can be measured in a number of ways [14]. We chose to test four simple definitions:
  - **1**: "one" is the constant number considered when a term appears at least once in the text. In this case the *tf* does not depend on the number of repeats of the term in the text.
  - **freq**: this is the frequency of the term in the text.
  - **freq/#ofTerms**: this option normalizes the frequency of the term by dividing it to the #ofTerms which is the length of the text (shown as *textLength*). Now, the *textLength* can be itself considered in two cases as mentioned below.
  - **log**: another normalization option that increases logarithmically with increases in the frequency of the term and is equal to $1 + \log_{10}(\text{freq})$.
- **textLength**: in case we need to count *#ofTerms* in a piece of text (e.g., the *freq/#ofTerms* option above), we have two options:
  - **before** (b): in this case, the length of the original text (i.e., the text before removing any other keyword that is not Stack Overflow tag) is measured as #ofTerms.
  - **after** (a): in this case, the length of the remaining text (that just includes Stack Overflow tags) is measured.
- **freq(t$_i$, q)**: we consider the four options similar to the *tf* mentioned above, except that this is measuring the counts of the words for the query *q*. Like *tf*, in its third option (*freq/#ofTerms*), the two cases for *textLength* will take effect as mentioned above.
- **prioritizing previous assignees**: As an enhancement in our overall assignment process, we can consider a priority for the developers who have fixed at least one bug before, over the rest of developers (i.e., at each point in time, first, rank the developers who have fixed at least one bug up to that point, then rank the other members in the developer community randomly (since they all have the score of *zero*). Note that this way it gives a chance to the developers who did not have a common keyword with the new bug and so obtained score of *zero*, but were active regarding other keywords, over the other developers who were completely inactive). The options are "yes" and "no". Note that the data set is not filtered in any case and at each point in time, only the previous evidence is used.

We used three projects for tuning. In order to select the projects for tuning, we divided the projects into three categories; projects with 1k-, 1k to 10k and 10k+ bug-assignments for small (4 projects), medium (4 projects) and large (5 projects) respectively. Then, we selected one project per category; we selected the smallest of each category for tuning: lift/framework, fog/fog and adobe/brackets. The idea was to leave the biggest projects with more bug assignments for the main experiment.

We used extensive 2d exploration [52,53] and performed a mutation experiment for tuning the parameters and obtaining the best configuration. We considered all the possibilities of the different parameters to create different configurations and obtained the results for the three test projects in each case. In total, there are 276 different cases for our 6 factors to tune the algorithm. So, we ran the program in a loop generating the above

276 cases. We obtained the results of each run, and sorted them (descending) based on MAP, calculated over all the bug reports of all the three projects. We looked at the top 10% in the results, and observed that we can decide for two of the parameters which we call *primary factors*. The *primary factors* are those that frequently appear with fixed value in those top 10% records and it is obvious that they have high effect on the output; *weighting* = "use term-weighting" and *recency* = "rel_recency". The rest of parameters ("TF", "prioritizing previous assignees", "freq(t_i, q)" and "textLength") are called *secondary factors* for which we cannot conclude anything at this step. For brevity, since the produced output contains too many results, we do not report the detailed results of this step. The 276 configurations are as follows:

(A) Primary factors; *Weighting* and *recency*:
Weighting has 2 cases ("do not use term-weighting" and "use term-weighting") and *recency* has 3 cases ("no recency", "abs_recency" and "rel_recency").
This makes:

2 (two cases of *weighting*) × 3 (three cases of *recency*) = **6 cases**.

(B) Secondary factors; $tf(t_i, sd_j)$, *prioritizing previous assignees*, *textLength* and *freq*(t_i, *q*):
$tf(t_i, sd_j)$ has four cases, and, in each case, '*prioritizing previous assignees* has two cases ("yes" and "no").
In three cases of $tf(t_i, sd_j)$ ("1", "freq" and "log"), for each case of *prioritizing previous assignees*, there are 5 cases for *freq*(t_i, *q*):

"1",
"freq",
"freq/#ofTerms" + *textLength*="before",
"freq/#ofTerms" + *textLength*="after" and
"log".

This makes:

3 (three out of four cases of $tf(t_i, sd_j)$) × 2 (two cases of *prioritizing previous assignees*) × 5 (five cases for *freq*(t_i, *q*)) = **30** cases.

In the other case of $tf(t_i, sd_j)$ (i.e., "freq/#ofTerms"), regarding each *prioritizing previous assignees*, there are 8 cases for *freq*(t_i, *q*) (four original cases for "freq", times two for the two cases of *textLength*; "before" and "after").
This also makes:

1 (the last case of $tf(t_i, sd_j)$) × 2 (2 cases of *prioritizing previous assignees*) × 8 (four cases for *freq*, times two for the two cases of *textLength*; "before" and "after") = **16** cases.

So for the secondary factors there are 30 + 16 = **46 cases**.

(C) Combining the above two sets (multiplying primary and secondary factors), in total, there are:

6 (primary cases) × 46 (secondary cases) = **276 cases**.

These cases took 2 hours in total to run on a machine with Core i7 CPU and 16GB of RAM. After that, we set the primary factors to those values obtained from the previous step (although we still do not know the exact effect of each primary factor and will examine it at the end of this step). Then, run the code with 46 unique cases including all possible combinations of the secondary factors. We obtained the best combination as shown in Table A1. The best configuration is shown in bold (and grayed cells) in the table.

It seems that the obtained MAP values are too close to each other. To check if there is a significant difference between the result of different configurations of secondary factors, we calculated the Coefficient of Variation (CV) of the MAP values mentioned in Table A1. The CV is too low (0.0005). This indicates that these (secondary) factors have only minor effect on the final MAP and they can be neglected. However, as a systematic way of 2d exploration[52,53], we choose the best result because it may make a higher difference when we run the code for the main

experiment (10 remaining projects) with several times more bug reports. The best results were obtained by setting the secondary factors as: "$tf(t_i, sd_j)$=freq/#ofTerms", "prioritizing previous assignees=yes", "$freq(t_i, q)$=freq" and "textLength=before(b)". By setting the secondary factors to those best obtained values, the effect of different combinations of primary factors as shown in Table 3 can be compared (12 cases as discussed in part "A" above).

TABLE A1 The results of tuning secondary factors affecting the accuracy of *VTBA* (46 cases in total). The best obtained configurations are highlighted.

$tf(t_i, sd_j)$	Prioritizing previous assignees	$freq(t_i, Q)$		MAP
1	yes	1		54.43
		freq		54.47
		freq/#ofTerms	tL = b	54.47
			tL = a	54.47
		log		54.42
	no	1		54.41
		freq		54.46
		freq/#ofTerms	tL = b	54.46
			tL = a	54.46
		log		54.40
freq	yes	1		54.43
		freq		54.47
		freq/#ofTerms	tL = b	54.47
			tL = a	54.47
		log		54.42
	no	1		54.41
		freq		54.46
		freq/#ofTerms	tL = b	54.46
			tL = a	54.46
		log		54.4
freq/#ofTerms	**yes**	1	tL = b	54.43
			tL = a	54.43
		freq	**tL = b**	**54.48**
			tL = a	54.47
		freq/#ofTerms	tL = b	54.47
			tL = a	54.47
		log	tL = b	54.42
			tL = a	54.42
	no	1	tL = b	54.41
			tL = a	54.41
		freq	tL = b	54.46
			tL = a	54.46
		freq/#ofTerms	tL = b	54.46
			tL = a	54.46
		log	tL = b	54.41
			tL = a	54.41
log	yes	1		54.43
		freq		54.47
		freq/#ofTerms	tL = b	54.47
			tL = a	54.47
		log		54.42
	no	1		54.41
		freq		54.46
		freq/#ofTerms	tL = b	54.46
			tL = a	54.46
		log		54.41