2016 7th International Workshop on Empirical Software Engineering in Practice

How to Provide Developers only with Relevant Information?

Stefan Räbiger, Ataman Girişken, and Cemal Yilmaz
Faculty of Engineering and Natural Sciences
Sabanci University
Istanbul, Turkey
Email: {stefan, ataman, cyilmaz}@sabanciuniv.edu

Abstract—**After the release of a new software version it is difficult for individual developers to keep track of all newly submitted bug reports complicating their decision making, e.g., which bug to resolve next? This problem is emphasized by the presence of further information sources, such as social media, which offer valuable user feedback to developers regarding the software. However, due to an abundant amount of information, developers might never notice this feedback. Hence, we envision a real-time system that provides developers with relevant information for improving the quality of their system while filtering out irrelevant facts from multiple information sources. For this system to work, it is necessary to compute the similarity between different types of documents, e.g., tweets and bug reports, in order to detect whether they are relevant to a developer or not. In this feasibility study, we focus on analyzing this core assumption in a simplified scenario in which we identify related bugs for a given software fix with the help of Natural Language Processing methods. In this experimental setting, which exhibits the key characteristics of our envisioned system, we obtain promising results indicating that our approach is feasible.**

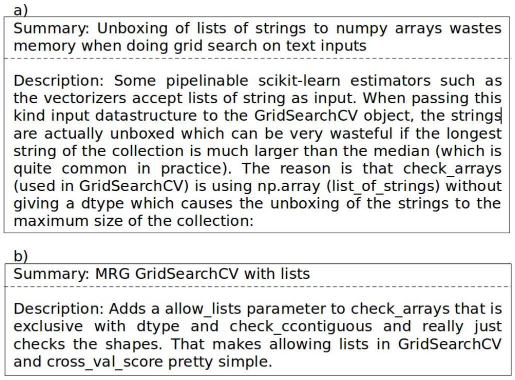
1. Introduction

Nowadays individuals are exposed to a wide range of information sources rendering them unable to distinguish relevant from irrelevant knowledge. This gives rise to the problem of information overload [1], i.e., not being able to find the relevant information due to the sheer amount of available data. Software developers are not exempt from that problem. Whenever a new piece of software is released, it is inevitable that new bugs are detected which in turn are reported by users, customers or developers themselves. At the end, these reports are added to the respective private or public bug repositories making it more and more difficult for individual developers to always keep track of all newly submitted bug reports, the progress state of bugs that are currently being resolved by other developers and so on. At times, this problem becomes even more severe, namely after releasing a new product or an improved version. Besides the bug repository, social media, such as Twitter or Facebook, have emerged over the last years as additional information sources for developers. For example, customers suggest improvements like “the button for closing the window should be in the top right corner” as comments below a Facebook post about a product or they tweet about encountered bugs, e.g., “upgrading my library from version 1.0 to 1.1 crashes my entire system”. However, developers notice such useful feedback rarely, mainly because of information overload.

Thus, we envision a real-time system that supports developers by filtering out unnecessary information and forwarding only “relevant” data. In this context, the definition of “relevant” depends on the information source. In the case of social media, any customer feedback could prove useful to developers, although developers might be currently working on different aspects of the program, but this type of user feedback helps for setting up long time visions and prioritizing new features. In contrast, the term implies for software repositories that only committed fixes or submitted bug reports are presented to a developer which are related to the piece of code on which she is currently working. For example, if she is addressing a specific bug and a user submits a new bug report related to the piece of code the developer works on, it should be displayed to her so that she may decide how to continue - either fixing the bug directly or leaving it open for further investigation. Similarly, our system can inform a developer A if someone else has just committed a fix that affects A’s current piece of code. Also the internal communication via email would be relevant to A if it addresses her code.

For this system to work, we must assume that all documents (bugs reports, fixes, tweets, Facebook posts, etc.) can be analyzed in a meaningful way, which allows to tell whether a document is relevant to a developer or not. Therefore, it is essential to verify the correctness of our hypothesis before implementing this system. To do so, we focus on the key concepts of this system, which involves comparing document similarity, retrieving only relevant information and all of those functions should be carried out in real-time or at least close to real-time. Hence, in this paper we study the scenario of finding related bugs for a fix that a developer is about to commit, i.e., she potentially entered additional comments to describe how the fix works. Bugs and fixes are comprised of a **summary** and a **description**. An example is illustrated in Figure 1. Note that we discard any source code regardless of whether it is part of a bug report or fix. Once a fix is about to get committed, our

978-1-5090-1851-2/16 $31.00 © 2016 IEEE
DOI 10.1109/IWESEP.2016.14



a)

Summary: Unboxing of lists of strings to numpy arrays wastes memory when doing grid search on text inputs

Description: Some pipelinable scikit-learn estimators such as the vectorizers accept lists of string as input. When passing this kind input datastructure to the GridSearchCV object, the strings are actually unboxed which can be very wasteful if the longest string of the collection is much larger than the median (which is quite common in practice). The reason is that check_arrays (used in GridSearchCV) is using np.array (list_of_strings) without giving a dtype which causes the unboxing of the strings to the maximum size of the collection:

b)

Summary: MRG GridSearchCV with lists

Description: Adds a allow_lists parameter to check_arrays that is exclusive with dtype and check_ccontiguous and really just checks the shapes. That makes allowing lists in GridSearchCV and cross_val_score pretty simple.

Figure 1. a) Sample bug report. b) The related fix.

system should display a list of likely related bugs in real-time which could be analyzed by the developer to decide whether one or more of the suggested bugs get resolved by the current fix as well. This concrete scenario covers all of the aforementioned key concepts of our future system because it has to work in real-time, a developer is forwarded only a relevant piece of information, namely that someone reported a bug related to the code on which this developer is currently working, and the similarity between different bugs and fixes must be calculated. In addition, recommending related bugs to a fix, which is about to be committed, has another advantage, namely easing maintenance of a bug repository. For example, if we assume that multiple bugs exist in a repository that are produced by the same root cause and if developers are working on fixing bugs with high priority first, it could be possible that bugs with lower severity could automatically get resolved when the latter are presented as part of the related bugs. It would also be possible that a related bug report was submitted recently and developers are unaware of it yet. Presenting such a bug report among the suggested related bugs would allow developers to resolve it directly as well.

We carry out a feasibility study to evaluate our basic hypothesis that calculating document similarity in this setting yields meaningful results in real-time. Here, bug reports as well as fixes represent documents which we compare with each other to find the k most similar bugs for a fix. To improve performance, we preprocess the documents with the help of Natural Language Processing (NLP) techniques before converting them into a vector space model to calculate pairwise similarities. To evaluate our approach, we resort to pairs of bug reports and fixes we collected manually from a real software repository. Each of the collected fixes addresses one specific report. The goal is to find the corresponding bug within the first k most similar bug reports calculated by our approach. Finding similarities between documents based on NLP techniques has been studied before in software engineering. However, to the best of our knowledge, we are the first ones to search for related bugs that could be resolved with a given fix.

The rest of the paper is organized as follows. Section 1 presents related work on measuring document similarities combined with NLP. Section 3 describes our approach to select the k most similar bugs; Section 4 explains our experimental design and we discuss our results. Section 5 explains potential threats to the validity; and Section 6 summarizes the proposed approach and discusses future work.

2. Related Work

The vector space model approach is widely used in information retrieval studies for identifying similar documents [2]. Vector space models work well on measuring word, phrase and document similarities. Most search engines are also using the vector space models to match queries with documents [3]. In our approach, we regard a single fix as a document as well as bug reports as separate documents to compute pairwise similarities for finding the most similar bugs for a given fix. While the aim in detecting duplicate bug reports is to find exact matches, we are only interested in similar matches, i.e., the output of our approach is a ranked list of potentially related bug reports and not a single bug report.

The use of Natural Language Processing (NLP) techniques is crucial in information retrieval problems in order to increase the recall rate which is the amount of retrieved relevant documents. These techniques include stop word removal and stemming in in the tokenization step. Stop word removal is to ignore common words that carry no information, e.g. ”the” or ”a”. Stemming is used to convert words into root words, e.g., “rains”, which could be the third person form of the verb “to rain” or the plural form of the noun “rain”. With the help of stemming, both forms would be transformed into “rain” which reduces the dimensionality in the vector model representation. This is important since it is well-known that this representation suffers from the curse of dimensionality, i.e., finding similarities across many dimensions tends to yield meaningless results. Therefore, reducing the dimensionality is of utmost importance for the vector space model to work well. However, other methods for dimensionality reduction could also be applied, commonly this would be latent semantic indexing [4], especially since stemming does not always lead to an enhanced performance of a system, e.g., tweets on Twitter tend to be short (at maximum 140 characters) and informal, so the heuristics applied for identifying root words do not always work. For example, in the work of Bao et al. [5], the authors report that using stemming affects their results negatively. Runeson et al. [6] calculate the textual similarity of bug reports to identify duplicates using NLP techniques, such as stop word removal and stemming. Afterwards, they convert those bug reports into a vector space representation which is used for computing pairwise bug report similarities to find the most similar ones. Finally, they present them to an end user to manually choose the duplicates from the presented list. Their approach is able to find an estimated maximum of 60% duplicates utilizing cosine similarity. Jalbert et al. [7] focus on finding duplicate bug reports in real-time in a bug report environment. In other words, previously unknown bug

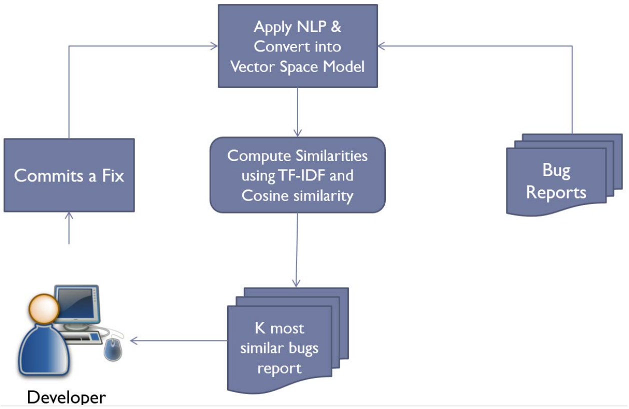

Apply NLP & Convert into Vector Space Model
Commits a Fix
Compute Similarities using TF-IDF and Cosine similarity
Bug Reports
K most similar bugs report
Developer

Figure 2. Overview of our approach to determine the k most similar bugs.

reports are matched with historical bug report data to reduce development costs. In a more recent study by Sun et al. [8], the authors focus on implementing a new similarity formula to increase the recall rate for finding duplicate bug reports. Again, in contrast to previous works we are not interested in finding duplicates, but rather related bugs to a given fix which could potentially also be repaired by that fix. The developer, who is about to commit her fix, has to decide whether any of the bug reports, suggested by our approach, get also resolved by that fix or not.

The underlying assumption for matching bug reports with their respective fixes, normally containing source code, is that both documents share common terms. Moreno et al. [9] provide empirical evidence that this assumption indeed holds. Hence, this work is closely related to our study. To compute similarities between source code and bug reports, the authors create a bag-of-words representation of both documents. Our work, in contrast, differs in three points. First, in our current implementation we disregard source code for matching bug reports with fixes - we focus on the summaries and descriptions for the sake of simplicity. Secondly, Moreno et al. do not utilize summaries or descriptions of fixes in their work. Thirdly, we represent our documents as TF-IDF vectors instead of using the bag-of-words approach.

3. Approach

An overview of our approach is depicted in Figure 2. We are given a set of bugs and a fix the developer is about to commit. After preprocessing the set of bugs, all bugs are converted into the vector space representation with TF-IDF weights and cosine similarity is employed to calculate pairwise similarities between the given fix and all bugs. The fix is converted into the same representation that was used for the bug reports. The list of the k most similar bugs is presented to the developer who must decide whether a suggested bug report gets resolved by her current fix or not. The details about this procedure follow in the next section.

3.1. How to Select the k most Similar Bugs?

For our proof of concept implementation, we utilize Scikit-learn [10] and NLTK [11]. From NLTK we employ the Snowball stemmer to obtain root words. First we tokenize bugs and fixes; we then preprocess them by stemming and lowercasing them, removing stop words and punctuation. The main reason for not performing more sophisticated strategies at the moment, e.g. removing adjectives and adverbs to reduce the dimensionality with the help of POS (Part of Speech) tags, is that we focus on investigating how well this approach is in general suitable to solve our problem at hand. More involved preprocessing strategies are to be performed in the future. We convert our bugs and fixes into a vector space representation to be able to compute inter-document similarities. Specifically, we employ TF-IDF [12] which is computed as follows:

$$w_{i,j} = tf_{i,j} \times log\left(\frac{N}{df_i}\right) \tag{1}$$

where $w_{i,j}$ is the weight of term i in document j, N is the number of documents in our collection, $tf_{i,j}$ is the term frequency of term i in document j and df_i is the document frequency of term i, i.e. in how many documents i occurs. Higher TF-IDF weights of words indicate that these words act as good discriminators for a document. Intuitively, weights are higher if words occur frequently in a document (the first term), but rarely in the document collection (the second term). In any other case, words are assumed to be not representative for a document resulting in lower weights. We expect TF-IDF to be a good representation for our problem since specific and seldom used function names occur in bug reports and fixes. For example, since function names could exist in various classes, normally the package of a function is also mentioned. After converting documents into the vector space representation, we compute the pairwise cosine similarity between fix d_j and bug d_k as:

$$sim(d_j, d_k) = \sum_{i=1}^{n} \frac{w_{i,j} w_{i,k}}{\sqrt{\sum_{i=1}^{N} w_{i,j}^2}\sqrt{\sum_{i=1}^{N} w_{i,k}^2}} \tag{2}$$

where $sim(d_j, d_k)$ lies between -1 (exactly opposite) and 1 (exactly the same), while a cosine similarity of 0 indicates no correlation at all. After computing the pairwise similarities of fix d_j with all bugs d_k, we store the k most similar bugs for d_j if $sim(d_j, d_k) > 0$.

4. Evaluation

With the help of our experiments, we want to analyze three research questions specifically related to bug reports and corresponding fixes:

1) Which parts of the documents should be compared to compute similarity?
2) Which preprocessing steps are useful for bug reports/fixes?
3) What is the execution time?

4.1. Dataset Description

We collected 46 matching pairs of bug reports and fixes from the Scikit-learn repository[1]. The main criterion for deciding whether to add a document pair or not was whether the fix linked to a bug report resolved the associated problem. If other participants in the respective bug thread reported further issues after applying the fix, we discarded the pair. Apart from that, we selected the documents randomly. As mentioned before, each of the documents is comprised of a summary and a description. Some of the bug reports were submitted by developers, while others were reported by users. During the collection of our dataset, we discarded any code snippets in the collected documents because our approach does not exploit this type of information yet. After stripping off the source code, in total 25 fixes and 4 bug reports did not include descriptions anymore.

4.2. Experimental Design

We simulate our scenario using those pairs, i.e., a developer has just finished adding her descriptions to the fix and she is about to commit it to the repository. The bug report that should initially be fixed using the commit is unknown to us, we instead utilize the matching bug report as the related bug that would automatically get resolved by committing the current fix. The reason for selecting this evaluation scheme is that the information necessary for knowing whether a fix resolves multiple bugs at a time or not is not available. Before starting with the experiments, the documents need to be transformed. To do so, at first a vector representation is generated based on all 46 bugs using Equation (1). In the next step, the fix is converted into the generated vector representation. Now its similarities with all bugs are computed according to Equation (2) and only the k most similar bug reports are retained.

4.3. Evaluation Framework

When looking at the k most similar bugs for a given fix, it is most important that the corresponding bug is contained in the list, ideally at the highest position in the ranking. Therefore, we use *recall* to evaluate our approach. More precisely, we utilize recall@k [13] as a metric which takes the recall up to rank k into consideration for the calculation. For example, when computing recall@3, the three most similar bugs returned for a given fix are checked - if the relevant bug is among those three bugs, it is counted as a match. This example also illustrates that the recall@k will always be either 1 or 0 in our experimental setup. Hence, we sum up the scores for all our 46 fixes to obtain our recall for a given k.

TABLE 1. DOCUMENT SIMILARITY USING DIFFERENT PARTS OF THE DOCUMENTS.

k	summaries	descriptions	descriptions & summaries
1	52.2%	19.6%	63.0%
2	58.7%	26.1%	71.7%
3	63.0%	26.1%	71.7%
4	65.2%	28.3%	76.1%

4.4. Which Parts to Consider for Document Similarity?

We analyze in this section the extent to which each part of a bug/fix is relevant for calculating pairwise document similarity. Since each document is comprised of two parts, we test all three combinations to quantify their importance: the documents could be described according to their a) summaries, b) descriptions, or c) a combination of the previous two options, i.e., both parts are merged into a single document. Calculating pairwise similarities between a fix and all bug reports leads to the results shown in Table 1. Considering summaries and descriptions of bugs and fixes as single documents yields the highest recall@k, while only performing pairwise comparisons based on the descriptions leads to a low recall. The latter observation is unsurprising since 29/46 documents feature no descriptions. But since source code is used regularly to illustrate bugs in software or to fix existing defects, we argue that our results are transferable to other repositories as well. On the one hand, using only summaries for computing document similarity works reasonably well, but on the other hand it depends on the fact that bug reports and fixes use the same words. Of course, the same problem exists when combining summaries and descriptions, but due to the larger amount of available text, this problem is less likely to occur. We note that it is unclear whether the good performance of sole summaries is an artifact of our dataset or not. To some extent they confirm the findings of Ko et al. [14] who noted that exploiting bug summaries only would suffice to identify similar documents as descriptions would be too noisy. However, at the same time our results contradict their findings because the performance benefits from descriptions. The results shown in Table 1 were obtained by tokenizing, lowercasing, and stemming all documents. We note that the same tendencies prevail when using different preprocessing methods (see Table 2 for an overview of all the techniques we tested), i.e., the combination of summaries and descriptions appears to be superior. Therefore, we opt for treating summaries and descriptions of a bug report/fix as single documents in the remainder of this paper.

4.5. Which Preprocessing Steps are Beneficial?

With this question we examine which preprocessing steps help achieve our goal. For example, employing stemming could lead to a deterioration of our results because this technique relies heavily on heuristics which do not hold true

[1] https://github.com/scikit-learn/scikit-learn

TABLE 2. EFFECT OF DIFFERENT PREPROCESSING STEPS ON DOCUMENT SIMILARITY.

k	T	T+L	T+L+S	T+L+P	All
1	58.7%	58.7%	63.0%	52.2%	56.5%
2	67.4%	67.4%	71.7%	65.2%	71.7%
3	69.5%	69.5%	71.7%	67.4%	71.7%
4	71.7%	71.7%	76.1%	76.1%	78.3%

for all datasets, e.g., the use of informal language makes it difficult to benefit from applying stemming to Twitter in specific tasks [5]. In general, stemming works better on longer texts and bug reports/fixes potentially vary heavily in length [15]. In our work, we focus on tokenization **(T)**, lowercasing **(L)**, stop word removal **(SR)**, stemming **(S)** and punctuation removal **P** as preprocessing techniques[2]. The effect of the different combinations of preprocessing steps on the recall@k is illustrated in Table 2. In the last column, indicated by “All”, we apply all previously mentioned techniques, that is T+L+P+SR+S. A combination of tokenizing, stemming and lowercasing documents seems most suitable to our problem. When increasing k, using all preprocessing methods becomes a viable alternative. Yet, the implications of choosing an appropriate k must be discussed. For it takes developers time to go through each suggested related bug and to think about whether her current fix resolves a listed bug, it becomes infeasible to present long lists. This problem is known as decision paralysis [16]. Due to the implications of this problem and based on Table 2, we opt for a small value of k, namely $k = 2$, as developers need to investigate at most two more bugs besides the one they are addressing with their current fix. The main reason is that the recall@k is not improving drastically thereafter for larger values of k. Since there is still room for improvement in terms of NLP techniques, e.g., POS tagging to remove adjectives or adverbs as they are not carrying meaning in this context, we find these results encouraging. If one would increase k, the results would naturally improve further. This would solely be due to the definition of recall@k. Suppose we set $k = 46$, then all 46 bug reports will be ranked in our list, so the recall@46 will be inevitably 100% regardless of our approach.

4.6. What is the Execution Time?

In the last question we study the execution time of our approach to show it performs document similarity calculations in real-time. The more preprocessing steps we combine, the longer it takes the system to compute the k most similar bugs for a fix. The results are illustrated in Table 3. Our system manages to yield results in at most 0.5 seconds, clearly showing the calculations are carried out almost in real-time. To compute the execution time, we averaged the results over 10 runs. We only depict them for $k = 1$ because the execution times remain the same for larger values of k as the same computations are performed in the background. For larger bug repositories the complexity of pairwise comparisons will increase linearly. The vector space model can be updated efficiently with the help of an incremental version of TF-IDF [17]. If the number of bug reports in the repository becomes too large, computing the similarities can be sped up by parallelization. Alternatively, it would be possible to cache similarities and recompute them only after every x instances, accepting some inaccuracies in the results.

TABLE 3. EXECUTION TIME IN SECONDS DEPENDING ON WHICH PREPROCESSING STEPS ARE APPLIED.

k	T	T+L	T+L+S	T+L+P	All
1	0.15	0.15	0.25	0.18	0.47

5. Threats to Validity

The dataset we used is small in size, meaning the results obtained using our approach might not be generalizable and the choice of the dataset could have affected our results as well. Another inherent threat is that the proposed approach completely relies on the assumption that similar words occur in similar bugs and respective fixes. Although it turned out that this observation held true for our dataset, we do not know whether other datasets would exhibit the same property. A way to alleviate this issue is incorporating more diverse sources for extracting information to compare the similarity of documents, e.g., source code. When discovering related bugs for a fix, those similar bugs can be resolved by changing the code in the vicinity of the given fix.

6. Concluding Remarks

In this paper we have explored the possibility to support developers actively by presenting them the most similar bugs to a fix those developers are about to commit. This allows them to realize that their fix is repairing multiple bugs at a time which simplifies the process of keeping track of ever-growing bug repositories as well as it is reducing the workload for bug triagers and developers. The preliminary results of our feasibility study suggest that the proposed approach is a promising one, but there is still room for improvement. We hope that the results of our feasibility lead to more research toward supporting developers in real-time while they are working on a piece of code.

In the future, we plan on making our approach more robust by incorporating source code snippets from descriptions of either bug reports or fixes. For example, in our work more than half of the documents contained source code that we discarded. That implies a good opportunity for improvement as pairwise similarity computations could turn out to be more accurate with these additional information. However, we would have to adjust our approach because

[2] We tried all combinations of these operations in preliminary experiments and due to space limitations, we only report the best ones.

NLP techniques do not work well on raw source code. One possible solution is described by Wang et al. [18], who extract class and method names from a bug report and build a vector space representation based on these information instead of using all words from a report. According to this approach the source code is parsed and all classes and methods (including overridden and overloaded ones), that were extracted from the bug report, are converted into a vector space representation. This allows to compute pairwise similarities and the most likely class/method would be considered to cause a bug. In this way bug reports and source code can be linked with each other. In the next step, we plan to retrieve costumer feedback from tweets for software developers. For example, the approach recently described in [19] utilizes NLP techniques and a bag-of-words representation to categorize short app reviews into four categories, including bug reports. This information would complement bug reports and prove useful to developers.

Since we have obtained promising results in this feasibility study, as an avenue for future research, we plan to extend our implementation for the system described in Section 1. To reduce the dimensionality of the TF-IDF vectors, we will utilize more advanced preprocessing methods like POS-tagging to be incorporated for reducing the dimensionality or the vector space model. For example, adjectives and adverbs usually do not carry information about bugs, thus removing them should not affect our results negatively. We also believe our described approach could be directly applied to bug repositories. In that scenario our approach would suggest a list of developers who have fixed bugs in the past similar to the one that was just submitted. This strategy could improve the efficiency of addressing bug reports automatically to assigning bug reports to developers.

References

[1] N. Davis, "Information overload, reloaded," *Bulletin of the American Society for Information Science and Technology*, vol. 37, no. 5, pp. 45–49, 2011.

[2] C. D. Manning and H. Schütze, *Foundations of statistical natural language processing*. MIT press, 1999.

[3] P. D. Turney and P. Pantel, "From frequency to meaning: Vector space models of semantics," *Journal of Artificial Intelligence Research*, vol. 37, pp. 141–188, 2010.

[4] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman, "Indexing by latent semantic analysis," *JAsIs*, vol. 41, no. 6, pp. 391–407, 1990.

[5] Y. Bao, C. Quan, L. Wang, and F. Ren, "The role of pre-processing in twitter sentiment analysis," in *Intelligent Computing Methodologies*, ser. Lecture Notes in Computer Science, D.-S. Huang, K.-H. Jo, and L. Wang, Eds. Springer International Publishing, 2014, vol. 8589, pp. 615–624.

[6] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," *International Conference on Software Engineering*, no. 29, pp. 499–510, 2007.

[7] N. Jalbert and W. Weimer, "Automated duplicate detection for bug tracking systems," in *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*. IEEE, 2008, pp. 52–61.

[8] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," *International Conference on Software Engineering*, pp. 253–262, 2011.

[9] L. Moreno, W. Bandara, S. Haiduc, and A. Marcus, "On the relationship between the vocabulary of bug reports and source code," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. IEEE, 2013, pp. 452–455.

[10] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *The Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[11] S. Bird, E. Klein, and E. Loper, *Natural Language Processing with Python*, 1st ed. O'Reilly Media, Inc., 2009.

[12] G. Salton, A. Wong, and C.-S. Yang, "A vector space model for automatic indexing," *Communications of the ACM*, vol. 18, no. 11, pp. 613–620, 1975.

[13] S. Büttcher, C. Clarke, and G. V. Cormack, *Information Retrieval: Implementing and Evaluating Search Engines*. The MIT Press, 2010.

[14] A. J. Ko, B. A. Myers, and D. H. Chau, "A linguistic analysis of how people describe software problems," in *Proceedings of the Visual Languages and Human-Centric Computing*. IEEE Computer Society, 2006, pp. 127–134.

[15] M. Kantrowitz, B. Mohit, and V. Mittal, "Stemming and its effects on tfidf ranking," *SIGIR conference on Research and development in information retrieval*, no. 23, pp. 357–359, 2010.

[16] G. Giddings, "Humans versus computers differences in their ability to absorb and process information for business decision purposesand the implications for the future," *Business Information Review*, vol. 25, no. 1, pp. 32–39, 2008.

[17] T. Brants, F. Chen, and A. Farahat, "A system for new event detection," in *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*. ACM, 2003, pp. 330–337.

[18] D. Wang, M. Lin, H. Zhang, and H. Hu, "Detect related bugs from source code using bug information," in *Computer Software and Applications Conference (COMPSAC), 2010 IEEE 34th Annual*. IEEE, 2010, pp. 228–237.

[19] W. Maalej and H. Nabil, "Bug report, feature request, or simply praise? on automatically classifying app reviews," in *Requirements Engineering Conference (RE), 2015 IEEE 23rd International*. IEEE, 2015, pp. 116–125.