RAPTOR: Release-Aware and Prioritized Bug-fixing Task assignment Optimization

Yutaro Kashiwa
Graduate School of Systems Engineering
Wakayama University
Wakayama, JAPAN
kashiwa.yutaro@g.wakayama-u.jp

***Abstract*—Over a decade, many bug assignment methods have been proposed in order to assist developers to read bug reports submitted daily and numerously, and to assign an appropriate developer. However, they tend to concentrate their assignments on a small number of particular developers. Applying the methods to the projects which have releases would reduce the number of bugs that developers can fix by the next release date because the time that developers can devote to bug-fixing is limited.**

In this study, we propose the release-aware bug-fixing task assignment method to mitigate the task concentration and increase the number of bugs that developers can fix by the next release date. This method employs mathematical programming to find the best combination at project level while the traditional methods find the best pair of a bug and a developer (at individual level).

***Index Terms*—Bug-triaging, Task assignments, Optimization, Software quality assurance, mining software repository**

I. Introduction

Software release cycle has to be shorter because developers are required to deliver new features or bug-fixes more rapidly than rival companies do in order to gain ground [1]. Moreover, software development is becoming more large-scale and more complicated, which embeds numerous bugs in their product [2]. Testing teams are able to find most of the bugs during test phases, yet their high number makes it difficult to fix them all by the next release because of lack of time. Core developers read bug reports, discuss, prioritize, and then assign them to appropriate developers. The work is well known for being tough due to the numerous daily submissions of bug reports [2], [3]. Therefore, many bug-fixing task assignment methods have been proposed for over a decade [2], [4]–[6].

Most of the methods collect fixed bug reports, extract and parse their description. The parsed text and the developer's name that brought the fix will be utilized for building a classifier (e.g., SVM [7]). When a new bug is reported, the methods will extract its description, use it as the input for the models, and then predict an appropriate developer.

However, they tend to concentrate their assignments on a small number of particular developers because the number of past bug fixes differs depending on the developer. This makes the training for each developer imbalanced. Generally, software development is tied to release dates, therefore the number of bugs that can be fixed by even experienced developers before each release remains limited. The task concentration on the specific developers by the traditional methods would end up reducing the number of bugs that they can fix by the next release date. Given that projects leverage the methods (especially, automated bug assignments), a release-aware method is needed to fix more bugs by the next release date, leveraging all the human and time resources of the project. We aim to improve the efficiency of bug-fixing activities **for projects** in totality, **NOT for individual developers** (as it has been done in previous studies). For the first step of this study, we address the following challenge,

Challenge I: Mitigating task assignments concentrating on specific developers

Furthermore, these methods do not take into consideration how important each bug-fix is, in other words, they regard all bugs as the same. Bugs range from those that do not highly disturb users and can be fixed by any developer (e.g., typos) to those that should be fixed immediately or require professional knowledge or advanced techniques in order to be resolved (e.g., crash [8] or security bugs [9]) [10]. Considering the impact on users or the difficulty, supports for decision making of which bugs should be prioritized and fixed by the release date are needed. Ideally, these methods assign **the more important bugs to the more highly experienced developers.** This study must also address the following challenge,

Challenge II: Assigning important bugs to experienced developers

In this study, we propose the Release-Aware And Prioritized Task Assignment Optimization Framework (RAPTOR) which moderates the bug-fixing loads for specific developers to increase the number of bugfixes by the next release date. We regard the bug-fixing task assignment problem as a combination problem about matching certain developers to certain bugs, and we formulate it as a multiple knapsack problem to find the optimal combinations. For the first challenge, we confirm that (1) RAPTOR mitigates the task concentration problem caused by existing methods and (2) assigns the appropriate amount of bugs so that developers fix more bugs by the next release date. For the second challenge, we focus on assigning the more important bugs to the more highly experienced developers.

2576-3148/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSME.2019.00101

II. Task Assignment Techniques

Over a decade, many bug assignment methods have been proposed to reduce the quantity of effort coming from assignments. We summarize bug assignment methods below.

Expertise-aware methods aim to assign bugs to the developer who has appropriate expertise which is calculated from similar bugs that developers previously fixed [4]. The similarity of bug reports is measured from the description present in the bug reports [2], [4], [6], [11]–[16] or source code history [17]–[20]. Anvik and Murphy build a classifier (e.g., Naive Bayes [21], SVM [7]) with the sets of words in the bug report and the name of the developer who fixed the bug [2]. The model can recommend developers who are capable of dealing with newly-reported bugs with relatively high accuracy (about 70%-75%).

Importance-aware methods consider the levels of priority or severity contained in bug reports which shows the importance of fixing the bug [22]–[24]. Priority and severity levels show how important it is to fix bugs for developer and users, respectively. Lin et al. have built a model considering priority and severity in addition to text data, which was used to conduct an empirical study with the data including Chinese characters and showed non-textual data is comparable to textual data [23].

Activeness-aware methods try to assign bugs to active developers in projects [6], [11], [25]. Wang et al. measured developers' activity scores in each component for a few months and built a method assigning bugs to the active developer who has the highest score in the component that the bug involves [25]. This method does not need training classifiers but has improved about 20% of the accuracy of assignments compared to expertise-aware techniques.

Experience-aware methods aim to assign bugs to developers who have contributed to projects [22], [26], [27]. Naguib et al. have proposed a method to rank developers based on the times of bug fixing activities (e.g., number of fixed bugs, number of comments, and so forth) [26]. The method achieved an average accuracy of 88% with the top 10 recommendation and outperformed the expertise-aware method [15].

Cost-aware methods aim to reduce bug fixing time, to keep the accuracy of assignments [5], [27], [28]. Park et al. have extended Anvik's method [2] and presented CosTriage which takes the cost of bug-fixing into consideration [5]. Costriage requires estimating the cost of the fixing time for each bug which is calculated on the average fixing time of similar bugs in addition to the possibility of which the assignment is appropriate, which is calculated in the same way as Anvik's method. Costriage assigns a bug to the most appropriate developer, based on the ratio of how accuracy is important compared to the quickness of bug-fixing which is determined beforehand. While the accuracy decreases by about 5% compared with Anvik's method, Costriage can reduce 7%-31% of the average bug-fixing time.

Release-aware methods aim to assign bugs so that the amount fixed by the next release date can be increased. Although the other methods (described above) have the advantages of recommending appropriate developers or reducing bug-fixing time, those methods tend to assign bugs to the few developers who most frequently fixed bugs [5], [28] (We will call this the "task concentration problem"). The number of bugs that even experienced developers can fix is limited because most projects have short release cycles and the time by the next release is limited, which should be taken into account when developing methods. To the best of our knowledge, there are no assignment methods that consider releases. In this study, we try to build a release-aware method in order to increase the number of bugs by the release. We place a limitation on the number of tasks which are assigned to each developer during a given period; the method assigns bugs under the constraint with considering the ability of developers.

III. RAPTOR: Release-Aware and Prioritized Task assignment Optimization fRamework

A. Overview of RAPTOR

In this study, we propose a Release-Aware and Prioritized Task assignment Optimization fRamework (RAPTOR). This optimizes the performance of the bug-fixing activity in projects to increase bug-fixes by the next releases. Regarding the concentration problem of traditional methods that was identified in the previous section, we propose imposing limitations on assigning bugs to each developer so that the developer can fix for a specific period. Under the constraints, RAPTOR finds **the best combination of bugs and developers** for the project while traditional methods recommend **the best pairs between a bug and a developer**. RAPTOR cannot always assign the bug to the best developer due to the constraint regarding the prevention of concentration.

To solve the combination problem, we regard the bug-fixing task assignment problem as a combination problem between bugs and developers and use mathematical programming. In the following section, we first describe a multiple knapsack problem, which is the closest to the bug-fixing task assignment problem, and we formulate the bug-fixing task assignment problem as the multiple knapsack problem.

B. Multiple knapsack problem

The multiple knapsack problem [29] [30] extends the well-known knapsack problem to multiple knapsacks. This is an optimization problem that finds the best combination of items (with certain weights and values) and knapsacks with maximum capacities so that the total value of the items in knapsacks is maximized. The multiple knapsack problem is formulated as follows.

$$Maximize: \quad \sum_{i=1}^{n}\sum_{j=1}^{m} v_i x_{ij} \tag{1}$$

$$Subject\ to: \quad \sum_{i=1}^{n} w_i x_{ij} \leq c_j \qquad (j = 1, 2, ..., m) \tag{2}$$

$$\sum_{j=1}^{m} x_{ij} \leq 1 \qquad (i = 1, 2, ..., n) \tag{3}$$

$$x_{ij} \in \{0, 1\} \tag{4}$$

where, v_i and w_i represent the value and weight of the i-th item, respectively, whereas x_{ij} is the objective variable, representing whether the i-th item is located in the j-th knapsack (x_{ij}=1) or not (x_{ij}=0).

The purpose of the multiple knapsack problem is to find combinations of x_{ij} values that maximize the equation 1 under the constraints of equations 2, 3, and 4. Equation 1 is the objective function and is used to determine whether one combination of objective variable values is better than the other. In this case, it aims to maximize the total value of the selected items. In contrast, equation 2 is a constraint that denotes that the total weight placed in the j-th knapsack must be less than the maximum capacity it can carry (c_j), and equation 3 prevents any item being placed in more than one knapsack. equation 4 denotes the constraint that the x_{ij} should only take values of 0 (not selected) or 1 (selected). These are able to be solved by a solver such as lp_solve [31].

C. Application for Task assignment problem

We formulate bug assignment as a multiple knapsack problem as follows.

$$Maximize: \quad \sum_{i=1}^{n}\sum_{j=1}^{m} P_{ij}x_{ij} \tag{5}$$

$$Subject\ to: \quad \sum_{i=1}^{n} C_{ij}x_{ij} \leq T_j \quad (j = 1, 2, ..., m) \tag{6}$$

$$\sum_{j=1}^{m} x_{ij} \leq 1 \quad (i = 1, 2, ..., n) \tag{7}$$

$$x_{ij} \in \{0, 1\} \tag{8}$$

The items and the knapsacks are replaced into bugs (B_i) and developers (D_j) respectively. The values and weights are replaced respectively into the suitability of developer j for fixing bug i (**preference** P_{ij}) and the estimated number of days it takes for developer j to fix bug i (**cost** C_{ij}). The preference is set up in different ways to tackle challenges I and II (in Section IV and V). The costs are calculated with the same procedure used in [5]. We categorize bugs with Latent Dirichlet Allocation [32], measure the median value of bug-fixing days by each category and developer, and use it as the cost.

The capacities of knapsacks are replaced into **available time slots** (T_j) for each developer (D_j), which is calculated from an **upper time limit** L_j of each developer set in advance and the total cost C_{ij} already assigned to developer D_j. Figure 1 shows the method of calculation of T_j.

The objective variables (x_{ij}) represent whether the bug (B_i) is assigned to the developer (D_j). We obtain a combination of items (bugs B_i) and knapsacks (developers D_j) that maximizes the objective (bug-fixing efficiency for the whole project) under each knapsack's weight constraint (maximum time available to each developer or **limit**).

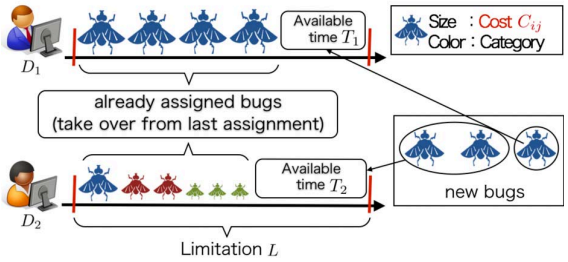

Fig. 1. Calculation of available time slot

IV. The Approach and Result of Challenge I

A. Approach

Settings: To evaluate the mitigation effect of RAPTOR, we set up **preferences** P_{ij} **in the objective function as the probability developer** D_j **can fix the bug** B_i, utilizing the same way as CBR and CosTriage. The probabilities are calculated by a classifier trained using the description of the bug and the name of fixers. Regarding the upper time limits L_j, we can prepare the limits for each developer but use the third quartile value of all the costs C_{ij}, which is a constant value, to simplify this study. The costs C_{ij} are calculated with the same procedure used in [5].

Datasets: We conducted a case study on three large OSS projects (Mozilla Firefox [33], the Eclipse Platform [34], and GNU compiler collection (GCC) [35]) which are commonly used in previous studies [4]–[6]. We prepare both learning and evaluation datasets. We use one year of data (from the first assignment day) as training data for all projects, 12 weeks of data (Firefox) and three months of data (Eclipse and GCC) before the main release as evaluation data. We execute the following three filters to validate data. Firstly, among all the collected bugs, we have only considered fixed bugs where the fixer and fixing time could be identified. Secondly, we removed bugs whose fixing-time meant they were outliers – Boxplots are used for this purpose. The reason for doing so is that some of the bugs were fixed after several years. Finally, we exclude the bugs fixed by not active developers to guarantee the accuracy of the task assignments [36]. We designate developers as "active", as those who had fixed six or more bugs within six months.

Experimental procedure: In this experiment, tasks were assigned using both the existing methods and RAPTOR, the bug-fixing times were calculated based on the resulting assignments. An overview of the experiment is shown in Fig. 2. Working through the experimental dataset (i.e., the evaluation data) in order, we extracted the bug reports for each date and used both the proposed and existing methods to assign the bugs. Once assignments had been made for all days, the bug-fixing times were calculated (Figure. 2, right). Since the assignment methods considered here do not always assign the bugs to the developers who fixed them, this means the actual bug-fixing time cannot be calculated. We used the

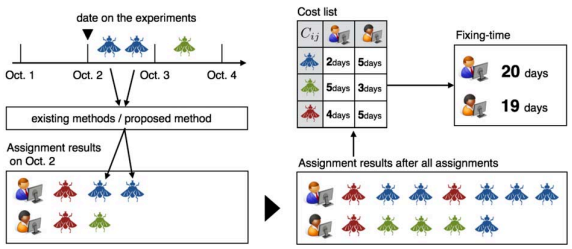

Fig. 2. Overview of experiment

median times taken by the individual developers to fix the bugs in each category (that is, the costs C_{ij}) from the training data as the bug-fixing times for the experiment.

Evaluations: We have prepared four evaluations to investigate whether RAPTOR could improve bug-fixing efficiency by assigning tasks to appropriate developers and considering the time they had available for bug-fixing, and compares the content-based recommendation (abbr. CBR) method [2] (expertise-aware) and CosTriage [5] (Cost-aware). We make sure whether RAPTOR can prevent tasks being concentrated on certain developers and can reduce the numbers of overdue bugs (which are assigned but fixed after the release). As an evaluation criterion of task concentration, the bug-fixing time should not exceed the evaluation data period for each project. Moreover, we evaluate RAPTOR in terms of the merit of the traditional methods and confirm the accuracy of assignments and bug-fixing. The accuracy of assignments measures a rate of the number of appropriate assignments and the number of all assignments. The appropriate assignment is defined as an assignment to the developer who has experienced fixing bugs with the same component as the target bug.

B. Result

Due to the limitation of space in this paper, we show the results of Firefox which are similar to the others.

[Finding 1] The traditional methods tend to concentrate tasks on some developers, but RAPTOR can mitigate the risk: We confirmed the number of tasks (the number of days to fix) that each active developer worked on. The number of developers who contributed for longer than the evaluation data period (Firefox: 12 weeks) using CBR and CosTriage, is eight and seven developers respectively. As for the task concentration by RAPTOR, the number of developers is four developers.

[Finding 2] RAPTOR can assign a more appropriate amount of bugs than the traditional methods do: We looked into the numbers of bugs that were assigned and fixed after the release. In Firefox, 77 bugs were overdue when using CBR, 75 bugs by CosTriage, and 31 bugs by RAPTOR. RAPTOR can assign a more appropriate number of bugs to each developer compared to the existing methods. On the other hands, there are 19 bugs that RAPTOR did not assign. Considering the un-assigned bugs, RAPTOR is still better than traditional methods, but this suggests that **RAPTOR should prioritize high priority bugs** (This is tackled in Challenge II).

[Finding 3] RAPTOR can reduce 46% of the bug-fixing time, compared with the traditional methods: The total bug-fixing days in Firefox is 1,843 days by CBR, 1,838 days by Costriage, 997 days by RAPTOR. RAPTOR could reduce about 46% of the days compared to both CBR and CosTriage.

[Finding 4] RAPTOR has a 21% lower assignment accuracy than CBR: The accuracy of assignments in Firefox was 81.7% by CBR, 81.0% by CosTriage, 60.6% by RAPTOR. This is because CBR is assigned to the developer with the largest preference, RAPTOR assigned bugs to developers so that the preferences are higher for the project (not for the developer). We are concerned that the low accuracy will induce a bug reassignment, and we are planning to investigate the negative impact of reducing accuracy. We will study methods to prevent from lowering it in future work.

V. The Plan of Challenge II (Curent State of Research)

In challenge I, RAPTOR mitigated the task concentration problem, but it has not taken into account bugs that should be prioritized and fixed by the release date. For challenge II, we plan to replace the preference P_{ij} into the following formulation designed for assigning more important bugs to more experienced and active developers.

$$P_{ij} = 10^{\,(pri_i \,*\, sev_i)} * sui_{ij} * \; con_{ij} * act_{ij} \tag{9}$$

where pri_i and sev_i represent the levels of priority and severity respectively. The priority and severity of Bugzilla [37] have five and seven levels respectively, we assign the value of 1 to the lowest level and gradually increment the numbers matching their severity and priority levels. To prioritize higher levels, we calculate the tenth power of the multiplied value of priority and severity scores.

sui_{ij} represents the possibility that the assignments of bug B_i is appropriate to developer D_j (the same way as Challenge I and the traditional methods).

con_{ij} is the number of contributions that D_j has fixed the same component of B_i. act_{ij} represents the activeness that D_j contributes to the component of B_i for the most recent period and refers to the inverse number of the days from the last date of contribution to the date when assigning bugs. These are set up for assigning bugs to well-experienced and active developers.

Multiplication of these parameters helps higher importance bugs to be assigned to experienced and active developers.

Acknowledgment

I would like to express my gratitude to my supervisor Masao Ohira, who is guiding me throughout my Ph.D. program. This research is conducted as part of Grant-in-Aid for Japan Society for the Promotion of Science Research Fellow and Scientific Research (JP17J03330).

REFERENCES

[1] F. Khomh, T. Dhaliwal, Y. Zou, and B. Adams, "Do faster releases improve software quality? An empirical case study of Mozilla Firefox," in *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, 2012, pp. 179–188.

[2] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 361–370.

[3] Y. Tian, D. Lo, X. Xia, and C. Sun, "Automated prediction of bug report priority using multi-factor analysis," *Empirical Software Engineering*, vol. 20, no. 5, pp. 1354–1383, 2015.

[4] J. Anvik and G. C. Murphy, "Reducing the Effort of Bug Report Triage: Recommenders for Development-Oriented Decisions," *ACM Transactions on Software Engineering and Methodology*, vol. 20, no. 3, pp. 1–35, 2011.

[5] J. Park, M. Lee, J. Kim, S. Hwang, and S. Kim, "COSTRIAGE : A Cost-Aware Triage Algorithm for Bug Reporting Systems," in *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, 2011, pp. 139–144.

[6] A. Tamrawi, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen, "Fuzzy set and cache-based approach for bug triaging," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 365–375.

[7] S. Gunn, "Support vector machines for classification and regression," University of Southampton, Faculty of Engineering, Science and Mathematics; School of Electronics and Computer Science, Tech. Rep., 1998.

[8] L. An, F. Khomh, and Y. G. Guéhéneuc, "An empirical study of crash-inducing commits in Mozilla Firefox," *Software Quality Journal*, vol. 26, no. 2, pp. 553–584, 2018.

[9] S. Zaman, B. Adams, and A. E. Hassan, "Security Versus Performance Bugs: A Case Study on Firefox," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, 2011, pp. 93–102.

[10] Y. Kashiwa, A. Ihara, and M. Ohira, "What Are the Perception Gaps Between FLOSS Developers and SE Researchers? A Case of Bug Finding Research," in *Proceedings of the 15th International Conference on Open Source Systems*, 2019, pp. 44–57.

[11] R. Shokripour, J. Anvik, Z. M. Kasirun, and S. Zamani, "A time-based approach to automatic bug report assignment," *Journal of Systems and Software*, vol. 102, no. C, pp. 109–122, 2015.

[12] S. Lee, M. Heo, C. Lee, M. Kim, and G. Jeong, "Applying deep learning based automatic bug triager to industrial projects," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 926–931.

[13] D. Cubrani and G. C. Murphy, "Automatic bug triage using text categorization," in *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*, 2004, pp. 92–97.

[14] X. Xia, D. Lo, X. Wang, and B. Zhou, "Dual analysis for recommending developers to resolve bugs," *Journal of Software: Evolution and Process*, vol. 27, no. 3, pp. 195–220, 2015.

[15] K. Somasundaram and G. C. Murphy, "Automatic categorization of bug reports using latent Dirichlet allocation," in *Proceedings of the 5th India Software Engineering Conference*, 2012, pp. 125–130.

[16] W. Wu, W. Zhang, Y. Yang, and Q. Wang, "DREX: Developer recommendation with K-nearest-neighbor search and EXpertise ranking," in *Proceedings of the 2011 18th Asia-Pacific Software Engineering Conference*, 2011, pp. 389–396.

[17] F. Servant and J. A. Jones, "WHOSEFAULT: Automatic Developer-to-Fault Assignment through Fault Localization," in *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 36–46.

[18] M. Linares-Vasquez, K. Hossen, H. Dang, H. Kagdi, M. Gethers, and D. Poshyvanyk, "Triaging incoming change requests: Bug or commit history, or code authorship?" in *Proceedings of the 2012 IEEE International Conference on Software Maintenance*, 2012, pp. 451–460.

[19] R. Shokripour, Z. M. Kasirun, S. Zamani, and J. Anvik, "Automatic bug assignment using information extraction methods," in *Proceedings of 2012 International Conference on Advanced Computer Science Applications and Technologies*, 2012, pp. 144–149.

[20] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. Hammad, "Assigning change requests to software developers," *Journal of software: Evolution and Process*, no. 1, pp. 3–33, 2012.

[21] G. John and P. Langley, "Estimating Continuous Distributions in Bayesian Classifiers," in *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, 1995, pp. 338–345.

[22] N. Kumar Nagwani and S. Verma, "Rank-Me: A Java Tool for Ranking Team Members in Software Bug Repositories," *Journal of Software Engineering and Applications*, vol. 05, no. 04, pp. 255–261, 2012.

[23] Z. Lin, F. Shu, Y. Yang, C. Hu, and Q. Wang, "An Empirical Study on Bug Assignment Automation Using Chinese Bug Data," in *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement*, 2009, pp. 451–455.

[24] M. Sharma, A. Tandon, M. Kumari, and V. B. Singh, "Reduction of Redundant Rules in Association Rule Mining-Based Bug Assignment," *International Journal of Reliability, Quality and Safety Engineering*, vol. 24, no. 06, p. 1740005, 2017.

[25] S. Wang, W. Zhang, and Q. Wang, "FixerCache: Unsupervised Caching Active Developers for Diverse Bug Triage," in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2014, pp. 1–10.

[26] H. Naguib, N. Narayan, B. Brügge, and D. Helal, "Bug report assignee recommendation using activity profiles," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, 2013, pp. 22–30.

[27] T. Zhang and B. Lee, "A hybrid bug triage algorithm for developer recommendation," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, 2013, pp. 1088–1094.

[28] J. Park, M. Lee, J. Kim, S. Hwang, and S. Kim, "Cost-aware triage ranking algorithms for bug reporting systems," *Knowledge and Information Systems*, vol. 48, no. 3, pp. 679–705, 2016.

[29] S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations*. New York: John Wiley & Sons, Inc., 1995.

[30] D. Pisinger, *Algorithms for Knapsack Problems*. Department of Computer Science, University of Copenhagen, 1995.

[31] "Lpsolve." [Online]. Available: http://lpsolve.sourceforge.net/5.5/

[32] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent Dirichlet Allocation," *Journal of Machine Learning Research*, vol. 3, pp. 993–1022, 2003.

[33] "Mozilla Firefox." [Online]. Available: https://www.mozilla.org/en-US/firefox/new/

[34] "Eclipse Platform." [Online]. Available: https://projects.eclipse.org/projects/eclipse.platform

[35] "GNU Compiler Collection." [Online]. Available: https://gcc.gnu.org/

[36] A. Mockus, R. T. Fielding, and J. D. Herbsleb, "Two case studies of open source software development: Apache and Mozilla," *ACM Transactions on Software Engineering and Methodology*, vol. 11, no. 3, pp. 309–346, 2002.

[37] "Bugzilla." [Online]. Available: https://www.bugzilla.org/