Poster: Leveraging Product Relationships to Generate Candidate Bugs for Duplicate Bug Prediction

Emily Su
Oracle Corporation
Redwood Shores, CA 94065, USA
emily.su@oracle.com

Sameer Joshi
Oracle Corporation
Redwood Shores, CA 94065, USA
sameer.joshi@oracle.com

ABSTRACT

Adaptive Bug Search (*ABS*) is a service developed by Oracle that uses machine learning to find potential duplicate bugs for a given input bug. ABS leverages the product and component relationships of existing duplicate bug pairs to limit the set of candidate bugs in which it searches for potential duplicates. In this paper, we discuss various approaches for selecting and refining the set of candidate bugs.

KEYWORDS

Duplicate Bug Prediction, Human Factors, Machine Learning

ACM Reference format:

Emily Su and Sameer Joshi. 2018. Poster: Leveraging Product Relationships to Generate Candidate Bugs for Duplicate Bug Prediction. In *Proceedings of 40th International Conference on Software Engineering Companion (ICSE '18 Companion). ACM, New York, NY, USA, 2 pages.* https://doi.org/10.1145/3183440.3195100

1 INTRODUCTION

Bug reports in Oracle help engineers track defects and maintain Oracle's products. Some of the bugs turn out to be duplicates, that is, two or more bugs with the same root cause. We developed Adaptive Bug Search (*ABS*), a service that identifies potential duplicates of a given input bug. Potential duplicates are the bugs most similar to the input bug. ABS first extracts features from bug reports such as subject, stack trace, etc. The machine learning model for ABS is trained with examples of duplicate and non-duplicate bug pairs after having computed pairwise similarity on their features [1,4]. ABS analyzes a new bug (*input bug*) by calculating its pairwise similarity with *candidate bugs*, applying a logistic regression model, and producing a report containing a list of up to *k* similar bugs. If the input bug is a duplicate bug, the report will ideally contain the *base bug*, the bug of which the input bug is a duplicate.

Oracle is a complex ecosystem with thousands of components in hundreds of products. Complex interdependencies exist between components of the same product, as well as components across different products [2]. When a bug reporter notices a symptom of a defect, s/he reports a bug in a specific *product* and *component*. At the time of reporting the bug, s/he may not have insight into the root cause, which may be in a different product and component [5]. A bug's component may not be updated, even after the bug is analyzed. We discuss how ABS accounts for this human behavior when identifying appropriate candidate bugs for a newly reported input bug.

2 NAIVE APPROACHES

We selected around 6,000 duplicate bug pairs reported in an Oracle product suite as data to test various approaches to determine the set of candidate bugs. We deem an input bug to be correctly predicted if its base bug appears anywhere in the report output by ABS. We chose k=8 for these experiments based on the number of bug reports users are willing to investigate. The *recall rate* [3] is calculated as the number of correctly predicted input bugs over the total number of input bugs.

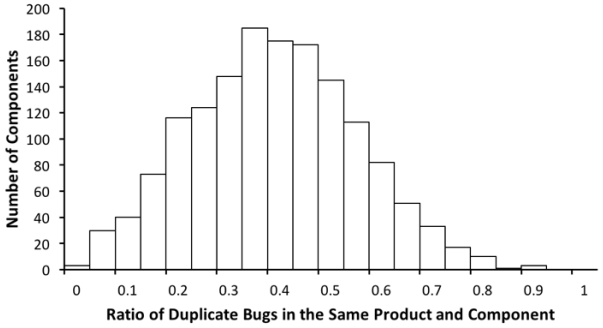

Figure 1: Number of Components vs. Ratio of Duplicate Bug Pairs with the Same Product and Component

ABS could consider selecting the candidate bugs from only the input bug's component (*Same Comp Only*). Table 1 shows that almost half the time, this approach fails to consider the base bug as a candidate bug because only 34.9% of duplicate bug pairs retrieved from 5 recent calendar years have matching products and components. To determine if mismatches are observed

ICSE '18 Companion, May 27-June 3 2018, Gothenburg, Sweden

https://doi.org/10.1145/3183440.3195100

across all components, we calculated the per component subset of these duplicate bug pairs with matching product and component values. In Fig. 1, we display results for components with at least 50 duplicate bug pairs.

In the next approach, *All Comps*, we expanded the set of candidate bugs to all components across all products. We find the base bug in the candidate bugs in 93.5% of the cases (Table 1). This value is not 100% because in all of the approaches, ABS filters out potential candidates based on the bugs' versions, fixed dates, operating system, etc. The *All Comps* approach is also undesirable because it selects several orders of magnitude more candidate bugs, leading to significantly slower run time. The recall rate for this approach was calculated using 1,000 bugs due to the very long run time. Run time is an important usability criterion for ABS since it is an interactive service.

Table 1: Approaches for Selecting Candidate Bugs

Approach	Med. Cand. Count	Ratio of Base Bug in Cand. Set	Recall Rate
Same Comp Only	2112	0.547	0.426
All Comps	1819321	0.935	0.625*
Candidate Comps	95886	0.935	0.637
Refined Candidate Comps	66531	0.919	0.631

3 CANDIDATE COMPONENTS APPROACHES

We devised a solution that takes advantage of many years of historical data. The data about existing duplicate bug pairs document inter-component relationships, which depend on product architecture and design. We generate a *related component set* for a given *input component*. A related component set is defined as the set of components associated with all the base bugs of all the existing duplicate bugs in the input component reported in the last 10 years. For example, as shown in Table 2, the related component set for the input component *X.k* is *{W.b, X.k, Y.m, Y.n, Y.o}*. Table 1 depicts that this approach, *Candidate Comps*, significantly reduces the median candidate count as compared to the *All Comps* approach and improves the recall rate as compared to both previous approaches. The recall rate is better than that of *All Comps* because this approach filters out similar candidate bugs in unrelated components. However, this approach tends to select too many candidate bugs because the candidate set includes a "long tail" of less related components with very few base bugs.

In a further refined approach, *Refined Candidate Comps*, we performed a benefit-cost analysis for each component in the candidate component set. The benefit/cost ratio for a given component is obtained by dividing the number of base bugs in this component by the total number of bugs reported in it. We construct the *refined candidate component set* by adding each candidate component in descending order of benefit/cost ratio, until we have included the components containing *<candidate threshold>*% of the base bugs for existing duplicate bugs in the input component. We performed a series of experiments, where we varied the candidate threshold and calculated the recall rate to obtain the optimal threshold (98%), the one that generates the best recall rate while still limiting the amount of candidate bugs. In the example in Table 2, we add components *Y.o*, *X.k*, *W.b*, and *Y.m*, in this order, such that we have included the components for 98% of the base bugs of existing duplicate bugs in input component *X.k*. The refined candidate component set for input component *X.k* is *{W.b, X.k, Y.m, Y.o}*.

Table 2: Example: Input Component X.k

Product.Comp	Base Bug Count	Total Bug Count	Benefit/Cost Ratio
W.a	0	N/A	0.000
W.b	120	6000	0.020
X.j	0	N/A	0.000
X.k	750	5000	0.150
Y.m	100	10000	0.010
Y.n	20	4000	0.005
Y.o	10	50	0.200
Z.p	0	N/A	0.000

4 CONCLUSION AND FUTURE WORK

In summary, we tried 4 different approaches for generating the set of candidate bugs for a given input bug. The ideal approach is a trade-off between having high recall rate and precisely limiting the number of candidates. The *Refined Candidate Comps* approach works well in practice since ABS has been running as a production service. However, this approach has some limitations. When new products and components are introduced, it takes time to build up enough product and component relationships for this approach to be effective. Even in the *All Comps* approach, we miss finding the base bug in the candidate set for 6.5% of the cases. One reason is that the heuristics for filtering the candidate bugs using the bugs' versions remain imperfect especially when comparing bugs in different products.

REFERENCES

[1] N. Jalbert and W. Weimer. 2008. Automated duplicate detection for bug tracking systems. In *Proceedings of International Conference on Dependable Systems and Networks With FTCS and DCC (DSN '08)*. IEEE, Anchorage, AK, USA, 52-61. DOI: 10.1109/DSN.2008.4630070

[2] Wanwangying Ma, Lin Chen, Xiangyu Zhang, Yuming Zhou, and Baowen Xu. 2017. How do developers fix cross-project correlated bugs?: a case study on the GitHub scientific python ecosystem. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 381-392. DOI: https://doi.org/10.1109/ICSE.2017.42

[3] Per Runeson, Magnus Alexandersson, and Oskar Nyholm. 2007. Detection of Duplicate Defect Reports Using Natural Language Processing. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. IEEE Computer Society, Washington, DC, USA, 499-510. DOI: http://dx.doi.org/10.1109/ICSE.2007.32

[4] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. 2008. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, New York, NY, USA, 461-470. DOI: http://dx.doi.org/10.1145/1368088.1368151

[5] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schroter and C. Weiss. 2010. What Makes a Good Bug Report? *IEEE Transactions on Software Engineering*, 36, 5 (Sept.-Oct. 2010), 618-643. DOI: http://dx.doi.org/10.1145/1453101.1453146