A Dataset of Non-Functional Bugs

Aida Radu
Department of Computing Science
University of Alberta, Edmonton, Canada
aradu@ualberta.ca

Sarah Nadi
Department of Computing Science
University of Alberta, Edmonton, Canada
nadi@ualberta.ca

***Abstract*—While several researchers have published bug data sets in the past, there has been less focus on bugs related to non-functional requirements. Non-functional requirements describe the quality attributes of a program. In this work, we introduce NFBugs, a data set of 133 non-functional bug fixes collected from 65 open-source projects written in Java and Python. NFBugs can be used to support code recommender systems focusing on non-functional properties.**

I. Introduction

a) Overview: Non-functional requirements (NFRs) are a class of software constraints that pertain to the quality of a program, as opposed to its functionality [1]. NFRs are an important part of software development, because they impact aspects of a program such as efficiency, portability, and maintainability. In this paper, we present NFBugs, a dataset of 133 real-world non-functional bugs from 65 open-source GitHub repositories. We define a *non-functional bug* as a problem in a piece of source code, which impairs some aspect of a program's NFRs. An example of a non-functional bug is when a developer uses an inefficient method when a more efficient one is available.

b) Previous Bug Datasets: While several researchers have published bug datasets in the past, none have focused primarily on non-functional bugs. MuBench by Amann et al. [2] contains 89 misuses of Java APIs, collected from existing bug datasets, open-source projects that used APIs from the `javax.crypto` package, and a developer survey. This set contains data from 33 open-source projects. Defects4J by Just et al. [3] contains 357 reproducible Java bugs collected from five open-source projects.

Bugs.jar by Saha et al. [4] contains 1,158 Java bugs from eight open-source Apache projects, constructed by extracting issue labels of type *Bug* from each project's commit history.

Ohira et al. [5] collected 4000 functional and non-functional bugs from four open-source Apache projects that had over 5000 reported issues in JIRA.

iBugs by Dallmeier and Zimmermann [6] contains 369 Java bugs from the AspectJ project. The authors created this dataset by searching for bug report IDs in the project's commit history. This dataset focuses on bugs that produce incorrect program results.

c) Novelty and Contributions: Most of the above datasets focus on functional bugs, and the ones that include non-functional bugs collect this data from less than 10 projects. While Ohira et al. [5] included non-functional bugs in their dataset, these were grouped broadly under the labels of either `security` or `performance`. Notably, our dataset uses 5 labels for non-functional bugs, in order to better differentiate the nuance of these bugs. Additionally, all the above datasets focus on Java projects. In contrast, our dataset contains NFR bugs from 65 projects written in either Java or Python. We also include a variety of projects from both novice and experienced programmers, so that their practices can be compared and contrasted. Most importantly, since our focus is on non-functional bugs, our dataset can be used to develop code recommender systems targeted at helping developers make better quality choices. NFBugs is publicly available on GitHub[1], under an MIT license with a citable DOI, and includes a simple API for querying the dataset.

II. Data Collection and Storage

To build NFBugs, we analyzed a variety of Github repositories written in either Java or Python. We used three methods to identify candidate repositories for mining. By examining the history of these repositories, we found commit messages related to non-functional bugs, which we then manually reviewed to construct our dataset.

A. Project Selection

1) Selecting Projects via Star Rating: Our first selection method uses the Github API to search for projects written in Java or Python, sorted based on their number of stars. We then analyzed the top repositories in order as follows.

To extract commits related to non-functional requirements, we implemented a keyword search for commit messages using PyDriller [7], which provides an API for mining git history. The stemmed keywords we use in our search are listed in Table I. Examples of possible word endings for matches are in square brackets. We constructed this set using a combination of the keywords created by Hindle et al. [8] and de la Mora and Nadi [9], as well as new keywords relevant to our focus. We also filtered out the terms `spell[ing]`, `typo`, and `npe` (null pointer error), since these kinds of fixes are irrelevant to this dataset.

Our PyDriller-based program mines the commit history of each project and identifies commits whose messages contain one or more of the keywords in Table I. It then outputs a CSV file containing the matched commit IDs and messages.

[1] https://github.com/ualberta-smr/NFBugs

TABLE I
COMMIT MESSAGE KEYWORDS

”fix”	”bug”	”error”
”refactor”	”secur[ity]”	”maint[enance]”
”stab[ility]”	”portab[ility]”	”efficien[cy]”
”usab[ility]”	”reliab[ility]”	”testab[ility]”
”changeab[ility]”	”replac[e]”	”memory”
”resource”	”runtime”	”crash”
”leak”	”attack”	”authenticat[ion]”
”authoriz[ation]”	”cipher”	”crack”
”decrypt”	”encrypt”	”vulnerab[ility]”
”minimiz[e]”	”optimiz[e]”	”slow”
”#”	”fast”	”perform[ance]

For projects written in Java, we further limited our search to commits affecting `.java` files. Likewise, for Python projects, we looked only at `.py` files.

We then manually reviewed these commits for true positives. We ignored commits that, despite containing one or more of the keywords in our list, did not relate to non-functional requirements (e.g., “Fix copy paste error”[2] and “Allow modules to replace visual character representation”[3]). Furthermore, we did not include problems that were highly specific to the given project. Using this first method, we reviewed the returned results from 91 Java and 15 Python repositories. A list of all our reviewed repositories is available in our online repository.

2) Selecting Projects via Github Search: Using our first method, we found several instances where developers replaced one API with another to improve performance. Noting these patterns, we developed another more effective strategy for finding potential target projects. In this second method, we identified candidate repos by using Github’s search bar to find relevant patterns. Github searches through a variety of data, including code, commit messages, and issues. We filtered our search to commit messages containing the keywords in Table I. As well, we searched for commit messages containing language-specific enhancements such as “StringBuilder replace,” and “foreach loop replace,” since we had seen these in previous projects. We analyzed the commit that was found in the search to confirm if it is relevant, then ran our PyDriller program on the complete repository history. Thus, these projects always had at least one potential hit. Once we ran PyDriller, the remaining steps were identical to those in the previous section. Using this method, we mined an additional 23 Java and 11 Python projects.

3) Selecting Projects from RepoReapers: In order to add more examples from well-engineered projects to our dataset, we chose Github repositories from the reaper dataset of well-engineered projects collected by Munaiah et al. [10]. First, we sorted this dataset in descending order of star rating. We then chose Java and Python repositories that the authors’ Random Forest classifier predicted as well engineered. We chose to focus on the predictions of the Random Forest classifier since Munaiah et al. reported a higher precision for that classifier, compared to the other classifiers in their study. We then ran our PyDriller-based program on these repositories as in the previous two methods. Using this dataset, we reviewed an additional 17 Java and 10 Python repositories.

B. Reviewing Potential Hits

For each identified potential hit (i.e., a commit that matched our criteria), the first author looked at the code changes that occurred in the commit to ensure the code change was indeed fixing a non-functional property. This manual review is necessary since there are sometimes matched commits that do not contain relevant code changes. For example, a commit message that matches a performance keyword may really be updating functionality[4], or it may confuse how APIs or methods were actually changed in the commit[5]. The second author then reviewed all confirmed and documented bugs. Any disagreements in labeling were discussed through issues or pull requests in the data set repository. As part of our reviewing criteria, we also ensured that we could both understand the fix and reasoning behind it before deciding to include it in the dataset. We erred on the side of precision.

C. Recording Identified Bugs

We categorized the bugs we found into one of three types: General-Practise, API-related, and Project-Specific. *General-Practise problems* are related to code quality and language-independent practices. *API-related problems* describe changes where one API or method was swapped with a “better” API. *Project-Specific problems* are non-functional changes that are specific to the logic of the project in which they occur; these typically include project-specific APIs.

For each problem we identified, we recorded the problem meta data in a YAML[6] file. Figure 1 shows an example of our format, which is similar to the format used by MUBench [2]. Each file specifies the problem’s `source`, which is the search method used to find this bug. The source `commit-msg-keywords` refers to method one, while `github-search` refers to method two, and `RepoReapers-dataset` refers to method three. Under the `project` field, we specify the `name` of the project and the `url` of its Github repository. We also document the `commit url` and `commit message` associated with the fix, as well as the particular `file` and `method` affected. Because a user’s commit message is not always thorough, we provide a `description` of our interpretation of the problem. In addition, we assign one or more tags to each problem according to the non-functional requirement(s) that the fix improves.

Our collection of tags consists of: `security`, `performance`, `memory`, `resource management`, and `determinism`. Problems tagged with `security` include changes such as improving encryption or preventing unauthorized usage. `performance` problems improve

[2]https://github.com/jenkinsci/jenkins/commit/9fb6ccf
[3]https://github.com/MovingBlocks/Terasology/commit/45f915a
[4]https://github.com/adangert/JoustMania/commit/a9711c2
[5]https://github.com/Offer-Ready/xslt-library/commit/12fbf12
[6]http://yaml.org/, last checked on June 27, 2018

```yaml
source:
    name: RepoReapers-dataset
project:
    name: elasticsearch
    url: https://github.com/elastic/elasticsearch
fix:
    tag: performance
    description: Replacing StringBuffer with StringBuilder improves performance because it is not synced
    commit message: >
            Use StringBuilder in favor of StringBuffer
            This removes all instances of StringBuffer that are removeable.
            Uncontended synchronization in Java is pretty cheap, but it's unnecessary.
    commit: https://github.com/elastic/elasticsearch/commit/1cf694b
location:
    file:
        core/src/main/java/org/elasticsearch/env/ShardLockObtainFailedException.java
        plugins/analysis-phonetic/src/main/java/org/elasticsearch/index/analysis/phonetic/Nysiis.java
    method:
        public String getMessage()
        public String nysiis(String str)
api:
    java.lang.StringBuffer
api change:
    java.lang.StringBuffer -> java.lang.StringBuilder
rule:
    if syncying is not required, use a StringBuilder instead of StringBuffer to handle strings efficiently
```

Fig. 1. Example YAML file for an API-related problem from `elasticsearch`

```yaml
source:
    name: RepoReapers-dataset
project:
    name: storio
    url: https://github.com/pushtorefresh/storio
fix:
    tag: performance
    description: Replacing enhanced for loops with regular ones improves performance because it saves memory space
    commit message: >
            Revert foreach for better performance in java
    commit: https://github.com/pushtorefresh/storio/commit/fbce95e
location:
    file:
        storio-common/src/main/java/com/pushtorefresh/storio/util/QueryUtil.java
    method:
        public static List<String> varargsToList('@Nullable' Object[] args)
suggestion:
    a for loop may be more efficient than a foreach loop
```

Fig. 2. Example YAML file for a General-Practise problem from `storio`

the program's time efficiency. `Memory` problems correct memory leaks or reduce memory usage. `Resource management` refers to problems where developers neglect to close a file or stream. Finally, problems tagged with `determinism` describe changes done to the program to ensure consistent behavior. For fixes related to specific APIs (e.g., `Java.lang.String`), we record the involved `API`. If the developer switched to a different API, we record the `API change` in the problem documentation (e.g., `Java.lang.String -> Java.lang.StringBuilder`). We also assign a `rule` to the problem, which is a generalized summary of the fix that other developers can apply. For commits not connected to a particular API, we instead create a general `suggestion` to address similar problems. An example of this type of record is shown in Figure 2.

III. Dataset Details

Using the first search method, 23 repositories contained true positives, out of the total of 91 Java and 15 Python repositories. In total, we identified 54 problems using this method. Using the second procedure, we obtained 35 problems from 18 Java repositories and 11 Python repositories. With the third procedure, we identified 44 problems from 8 Java and 5 Python repositories. In total, our dataset contains 133 problems from 65 projects. The distribution of problems by type and language is shown in Table II.

TABLE II
Data Distribution by Problem Type and Language

Problem Type	Python	Java	Total
API Related	24	68	92
General	16	16	32
Project Specific	3	6	9
Total	43	90	133

TABLE III
Data Distribution by Problem Tag

Tag	Python	Java	Total
security	10	15	25
performance	27	36	63
memory	3	33	36
resource management	2	14	16
determinism	1	2	3

The number of problems per tag are displayed in Table III. Performance related bugs were the most common subtype. Table IV shows the number of projects in our dataset that have stars, watches, and forks within a given range. Eighteen of the projects had no stars; all of these were identified via the `github-search` procedure. We decided to keep these projects in the dataset, because (1) some of these problems resembled others in more popular repositories and (2) it may interest others to study the non-functional bugs that novice programmers run into and fix. Meanwhile, 49 projects had at least one star, with the detailed breakdown shown in Table IV. Twenty-one projects had more than 800 stars. The majority of projects in our dataset had less than fifty watches and no more than 400 forks.

We identified 48 different Java APIs and 17 different Python APIs as sources of API-related problems. Fixes for these problems consisted of refining the use of the problematic API, or exchanging it with a more optimal one. The number of problems involving each API in each language is shown in Tables V and VI. Java's String class contributed to the highest number of API-related problems in that language. These were all `performance` problems, which developers fixed by using a more efficient API (e.g., StringBuilder) to perform concatenation. In Python, list methods were the most common source of API-related problems.

To enable easy usage of the dataset, we supply a `DataBox` Python API that allows users to filter the data based on star rating, tags, project source, and more. We also provide a sample client script[7] to demonstrate usage of the API.

[7]https://github.com/ualberta-smr/NFBugs/blob/master/scripts/DataBoxClient.py

TABLE V
FREQUENCY OF PROBLEMATIC JAVA APIS

API	Occurrence	API	Occurrence
java.lang.String	8	java.io.BufferedOutputStream	1
android.content.Context	2	org.apache.commons.io.IOUtils	1
java.io.FileOutputStream	2	hudson.model.Run	1
java.util.jar	1	hudson.model.Item	1
java.io.BufferedReader	3	hudson.model.Hudson	3
java.util.zip.ZipFile	1	org.kohsuke.stapler.StaplerRequest	1
java.util.ArrayList	2	java.io.FileReader	1
android.util.SparseArray	1	squareup.haha.perflib.Snapshot	1
java.util.Vector	3	java.lang.Long	3
java.lang.StringBuffer	5	java.lang.Double	2
SentienceLab.PointCloudDatasetReader.DataSource	1	java.lang.Boolean	1
xtremelabs.robolectric.bytecode.RobolectricClassLoader	1	java.lang.ref.WeakReference	2
org.codehaus.jackson.JsonParser	1	java.util.HashMap	3
fasterxml.jackson.annotation.JsonProperty.Access	2	java.lang.annotation.RetentionPolicy	1
terasology.rendering.nui.NUIManager	1	java.util.UUID.randomUUID	1
terasology.physics.engine.PhysicsLiquidWrapper	1	hudson.remoting.Channel	1
terasology.entitySystem.entity.EntityRef	1	android.webkit.WebView	1
java.io.FileInputStream	1	java.awt.image.BufferedImage	1
java.util.HashSet	1	java.io.Closeable	1
javax.imageio.ImageWriter	1	android.webkit.WebSettings	1
com.facebook.common.references.CloseableReference	1	java.lang.Float	2
java.util.concurrent.Executors	1	rajawali.util.LittleEndianDataInputStream	1
javax.imageio.ImageReader	1	org.apache.lucene.store.IndexOutput	1
PowerManager.WakeLock	1	java.io.ByteArrayOutputStream	3

TABLE VI
FREQUENCY OF PROBLEMATIC PYTHON APIS

API	Occurrence	API	Occurrence
builtins.io.IOBase	1	builtins.list	5
builtins.input	1	builtins.forloop	2
builtins.str	3	builtins.map	3
django.db.models.Model	1	builtins.hasattr	1
Object.__class__.__name__	1	requests	1
sqlite3.Connection	1	numpy	1
copy.deepcopy	1	builtins.generator	1
Lib.json	1		

TABLE IV
NUMBER OF PROJECTS BY GITHUB STATS

Range	Projects in Range		
	stars	watches	forks
[0]	17	5	20
[1,50)	9	37	10
[50,200)	2	14	16
[200,400)	6	4	4
[400,600)	6	1	5
[600,800)	4	0	2
[800,∞)	21	4	8

IV. DISCUSSION

Because of the manual labor needed for the review of keyword hits, we are only able to include a limited number of problems in our dataset. Additionally, similar to any manual review process, our decision to include a bug or not and its categorization is subjective. We mitigitated this by having two authors review all data and documenting all the steps taken to create the dataset. Additionally, by making the repository public and providing a template for documentation of bugs, we leave the dataset open to future extensions, whether by ourselves or by other researchers.

Having a dataset of non-functional bugs can enable the creation of code recommender systems that suggest potential API replacements to developers. In the future, we can also test how useful such a non-functional recommender tool is, and how important these kinds of bugs are to programmers. Careful thought needs to go into how such recommendations would be presented to the developer and when, since the detected problems are not functional defects in the end, but are potential improvements to the non-functional qualities of the system. Our data set can serve as a benchmark for such recommender systems. The data we collected can also be used in software documentation systems [11], to record non-functional problems with certain APIs. Additionally, our data can be used to expand the findings of previous work related to profiling performance bugs. Finally, since our dataset contains non-functional bugs from Java and Python, the types of non-functional problems found across the programming languages can also be compared.

V. CONCLUSIONS

This paper introduces NFBugs, a database of Java and Python bugs related to non-functional requirements. Currently, the dataset contains a description of 133 non-functional bugs collected from 65 open-source projects. The collected bugs come from a variety of projects and can be filtered using several attributes. The data can be used for studying types of non-functional problems in practice and for evaluating code recommender systems.

ACKNOWLEDGEMENTS

This work was sponsored by the Natural Sciences and Engineering Research Council (NSERC) Undergraduate Student Research Award (USRA), and by the University of Alberta's Computing Science Department Science Internship Program (SIP).

REFERENCES

[1] G. Kotonya and I. Sommerville, *Requirements Engineering: Processes and Techniques*, 1st ed. Wiley Publishing, 1998.

[2] S. Amann, S. Nadi, H. A. Nguyen, T. N. Nguyen, and M. Mezini, "Mubench: A benchmark for api-misuse detectors," in *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16. New York, NY, USA: ACM, 2016, pp. 464–467. [Online]. Available: http://doi.acm.org/10.1145/2901739.2903506

[3] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 437–440. [Online]. Available: http://doi.acm.org/10.1145/2610384.2628055

[4] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. R. Prasad, "Bugs.jar: A large-scale, diverse dataset of real-world java bugs," in *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18. New York, NY, USA: ACM, 2018, pp. 10–13. [Online]. Available: http://doi.acm.org/10.1145/3196398.3196473

[5] M. Ohira, Y. Kashiwa, Y. Yamatani, H. Yoshiyuki, Y. Maeda, N. Limsettho, K. Fujino, H. Hata, A. Ihara, and K. Matsumoto, "A dataset of high impact bugs: Manually-classified issue reports," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 518–521. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820518.2820602

[6] V. Dallmeier and T. Zimmermann, "Extraction of bug localization benchmarks from history," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07. New York, NY, USA: ACM, 2007, pp. 433–436. [Online]. Available: http://doi.acm.org/10.1145/1321631.1321702

[7] D. Spadini, M. Aniche, and A. Bacchelli, *PyDriller: Python Framework for Mining Software Repositories*, 2018.

[8] A. Hindle, N. A. Ernst, M. W. Godfrey, and J. Mylopoulos, "Automated topic naming to support cross-project analysis of software maintenance activities," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11. New York, NY, USA: ACM, 2011, pp. 163–172. [Online]. Available: http://doi.acm.org/10.1145/1985441.1985466

[9] F. L. de la Mora and S. Nadi, "Which library should i use? a metric-based comparison of software libraries," in *Proceedings of the 40th International Conference on Software Engineering New Ideas and Emerging Results Track (ICSE NIER '18)*, 2018.

[10] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating github for engineered software projects," *Empirical Software Engineering*, vol. 22, no. 6, pp. 3219–3253, Dec 2017. [Online]. Available: https://doi.org/10.1007/s10664-017-9512-6

[11] M. P. Robillard, A. Marcus, C. Treude, G. Bavota, O. Chaparro, N. Ernst, M. A. Gerosa, M. Godfrey, M. Lanza, M. Linares-Vsquez, G. C. Murphy, L. Moreno, D. Shepherd, and E. Wong, "On-demand developer documentation," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2017, pp. 479–483.