



Yet Another Miner Utility Unveiling a Dataset: CodeGrain

Dániel Horváth¹ ^a and László Vidács^{1,2} ^b

¹Department of Software Engineering, University of Szeged, Dugonics tér 13., Szeged, Hungary

²MTA-SZTE Research Group on Artificial Intelligence, Hungary

Keywords: Automated Program Repair, Data Mining, Bug-Fixing.

Abstract: Automated program repair (APR) gained more and more attention over the years, both from an academic, and an industrial point of view. The overall goal of APR is to reduce the cost of development and maintenance, by automagically finding and fixing common bugs, typos, or errors in code. A successful, and highly researched approach is to use deep-learning (DL) techniques to accomplish this task. DL methods are known to be very data-hungry, but despite this, data that is readily available online is hard to find, which poses a challenge to the development of such solutions. In this paper, we address this issue by providing a new dataset consisting of 371,483 code examples on bug-fixing, while also introducing a method that other researchers could use as a feature in their mining software. We extracted code from 5,273 different repositories and 250,090 different commits. Our work contributes to related research by providing a publicly accessible dataset, which DL models could be trained, or fine-tuned on, and a method that easily integrates with almost any code mining tool, as a language-independent feature that gives more granular choices when extracting code parts from a specific bugfix commit. The dataset also includes the summary, and message of the commits in the training data which consists of multiple programming languages, including C, C++, Java, JavaScript, and Python.

1 INTRODUCTION


Data-driven repair approaches (Monperrus, 2020) have recently shown promising results (Chen et al., 2019; Dinella et al., 2020; Lutellier et al., 2019; Jiang et al., 2021; Yi et al., 2020) in software engineering research. However, the most commonly used datasets for APR research (Defects4J (Just et al., 2014), QuixBugs (Ye et al., 2019), ManyBugs (Le Goues et al., 2015)) have some limitations, such as being too small and not preprocessed for deep learning.


To tackle this problem, Tufano *et al.* (Tufano et al., 2019a) introduced a large dataset mined from GitHub (GitHub, 2023a) commits that has become a benchmark (Lu et al., 2021) for various tasks. However, this dataset only considers learning on Java code, which limits researchers' ability to test configurations in a multilingual environment and poses significant limitations to research questions such as the ability to transfer knowledge (Kim et al., 2022) learned in one programming language to another. Also, the dataset uses an abstracted code format (Tu-

fano et al., 2019a; Csuvik and Vidács, 2022). However, current DL models can handle large vocabularies, which may motivate the use of non-abstracted code so that models can produce usable code as-is.

More recent work such as PreciseBugCollector (He et al., 2023) or GitBug-Actions (Saavedra et al., 2023) employ more sophisticated methods on bugfix collection. However many seminal works (Tufano et al., 2019a; Lutellier et al., 2020; Chen et al., 2019; Drain et al., 2021) use simple text-based verification of whether a commit is considered a bugfix or not. While more sophisticated methods could yield better quality bugfixes, it also limits one's ability for large-scale mining, since it greatly restricts bugfix commit availability, *e.g.* only commits that rely on relatively new features (such as github-actions) are considered for mining.

Another issue with many of the datasets available is that they store bugfixes in raw format, without the employment of some sort of pre-processing, such as extracting smaller, but complete segments of the code, like methods or classes. CodegrainHouse separates such code fragments from the rest of the code. However, if one should need it, diff information is also available in the dataset which contains the modifications applied to the entire source file.

^a  <https://orcid.org/0000-0001-8855-921X>

^b  <https://orcid.org/0000-0002-0319-3915>

Without further ado, we propose *CodeSieve* utility, and *CodegrainHouse*, a newly mined dataset featuring our technique. *CodegrainHouse* is a multi-linguistic dataset consisting of 5 popular programming languages, mined from more than 5,000 GitHub repositories. It contains 371,483 methods extracted from code from 250,090 bug-fixing commits. The dataset also contains the message of the commit, allowing researchers to not only generate the fixed code but include an explanation on the generated fix, which could motivate researchers to make DL models to also generate commit messages (Xu et al., 2019) when producing a fix, which also helps the process of automation and documentation. *CodegrainHouse* is larger and more detailed compared to similar datasets publicly available on code repair. The dataset is available at Zenodo¹ in a JSONL format, which simply contains JSON-compatible strings separated by line breaks. Our methods used for extracting functions from commit diffs are also available as a small package at our public GitHub repository².

2 RELATED WORK

In this section, we provide some of the related work that collected datasets containing bugs, or bugfixes. These datasets are usually used by G&V approaches. Some of which can be used in test-based, and some in deep-learning contexts.

Defects4J (Just et al., 2014), *BugsJS* (Gyimesi et al., 2019), *BugsInPy* (Widyasari et al., 2020) are similar benchmarks, for Java, JavaScript, and Python respectively. The datasets consist of a relatively small number of, although high-quality, manually curated code examples on bugfixes.

Codeflaws (Tan et al., 2017) benchmark dataset contains 7,436 programs sourced from the Codeforces online database. This dataset includes 3902 defects categorized into 40 different groups of defects. As a result, it allows for researchers to apply, and test a wide range of repair tools on these bugs.

QuixBugs (Ye et al., 2019) consists of 40 programs from the Quixey Challenge translated into both Python and Java. It is a defect collection of one-liner bugs, with corresponding test cases. Bugs in this dataset can be categorized into 14 different classes. The benchmark can be used to study the performance of multi-language repair tools.

ManySStuBs4J (Karampatsis and Sutton, 2020) is a collection of simple Java bug fixes for evaluating

program repair techniques. They also use specific keywords such as *error*, *issue*, *fix*, *repair*, *solve*, *remove*, *problem* to identify bug-fixing changes. They also filter them to get small bugfix changes and categorize them into 16 syntactic templates called SStuBs. The dataset includes bugs from open-source Java projects mined from GitHub, focusing on single statement changes that are likely bug fixes, similar to our work in the sense that we also only include commits affecting a single file.

Vul4J (Bui et al., 2022) is a dataset of 79 reproducible Java vulnerabilities from 51 open-source projects filtered from the Project KB (Ponta et al., 2019) knowledge base. Each vulnerability comes with a proof of vulnerability (PoV) test case. They collected and analyzed 1803 fix commits from 912 real-world vulnerabilities to create the dataset.

FixJS (Csuvik and Vidács, 2022) similar to the Java benchmark introduced by Tufano *et al.* (Tufano et al., 2019a) usually used by papers found in CodeXGLUE (Lu et al., 2021), but for JavaScript bugs. Similarly to our work, it uses commit messages for filtering bugs from GitHub repositories.

GitBugActions (Saavedra et al., 2023) is a tool for building new datasets, that rely on GitHub Actions to detect and mine bugfixes including reproducible test suites. The main focus of the tool is bugs that have corresponding test suites which can be collected by locally executing them in the environment specified by the developers in the GitHub Actions workflow.

PreciseBugCollector (He et al., 2023) is a dataset focusing on bugfix quality. They employ a precise bug collection approach comprising of a bug-tracker and a bug-injector component. They extracted 1,057,818 bugs from 2,968 open-source projects, where 12,602 bugs come directly from bug repositories (such as NVD and OSS-Fuzz), while the remaining 1,045,216 bugs are injected via bug-injection tools.

CodegrainHouse provides a relatively large dataset containing smaller, more separated code parts (*e.g.* functions) to be fixed. To the best of our knowledge, this is yet, the largest dataset to not only provide the buggy and fixed code, but also the corresponding commit message, commit diff, project's name, the fixed file's location in the repository, the current, and parent commit's hash, and more.

3 BACKGROUND

Software development is a complex process that involves several stages, from writing code to delivering the final product to customers. For developers, ver-

¹<https://zenodo.org/records/10198721>

²<https://github.com/AAI-USZ/codesieve>

sion control systems like git, play a vital role in developing the final product.

When developers complete a feature or fix a bug, they commit code changes to the version control system. Committing code involves creating a snapshot of the changes made to the codebase. This snapshot –among various other information– includes the changes made to the code and a description of the changes. The committed code is then pushed to a remote repository, *e.g.* GitHub, where other team members can access it.

We make use of this process by querying the GitHub API (GitHub REST API, 2023a) to search for desirable repositories. We then copy them to a local store to iterate the commits and filter them for bug-fixing commits. Datasets like GitBugActions (Saavedra et al., 2023), PreciseBugCollector (He et al., 2023) provide us with great bug collection methods, FixJS (Csuvik and Vidács, 2022) can give us more fine-grained code snippets as it only contains functions, and not necessarily the whole source file, QuixBugs (Ye et al., 2019) or Defects4J (Just et al., 2014) have few, although, high-quality collection of defects with corresponding test cases. However, these datasets have either too few examples for deep learning models to train effectively, provide only a single language, or do not give control over which parts of the code the user is interested in case of a bugfix. To this end, we introduce a method that could be used as a feature in data mining software independent of programming language and also provide a collection of relatively up-to-date bug-fixing commits in our dataset.

4 METHOD

In this section, we will explain the mining process. This task involves multiple steps necessary to extract the desired code snippets from the source code. First, we query GitHub’s API to find the repositories that meet our filtering criteria. Once we have located these repositories, we clone them to a specified location on our system. After that, we iterate through each repository while filtering for bug-fixing commits. Finally, we extract the desired code snippets from the source code and save them to a specified location. This step is important since we are only interested in code snippets that are related to bug fixes.

4.1 Mining

Online repositories like GitHub (GitHub, 2023b) are an abundant resource of source codes. During

the mining process, we used GitHub’s API (GitHub REST API, 2023a) to query for appropriate repositories. The only API endpoint required by this approach is to use the *repositories* endpoint <https://api.github.com/search/repositories> with the appropriate query parameters. Among other things, a high-level approach to this repository filtering process can be observed in Figure 1. The public API is only used in the first part of the process, where repositories are queried based on some criteria, *e.g.* the publicity, number of stars, *etc.*

After collecting necessary information on repositories, like the URL, the cloning process begins. For the cloning process, we used GitPython (GitPython, 2023), a popular Python package for interacting with git repositories. We selected more than 5,000 repositories based on the following criteria; the repositories should be public, and should not be mirrors, forks, templates, or archived. Furthermore, at the time of the collection process, there should be a push event not older than 180 days, signaling that the project is still being maintained.

We collected source codes written in 5 popular programming languages, C, C++, Java, JavaScript, and Python. These languages are also included in the API query criteria.

After the cloning process, one is able to freely iterate the commits without the use of GitHub’s API, which we did try to minimize as much as possible throughout our experiments, and the development of the data creation software. One reason to do so is because the public API can be limiting (GitHub REST API, 2023b) in the sense, that it only allows a small number of queries per hour. Specific commits are primarily selected by their commit message. If the message contains a “fix-like” word, such as “bug”, “fix”, or “improve” (Tufano et al., 2019b), then the commit is considered a bugfix. For our purposes, the commit should also conform to 3 other rules.

Firstly, there should be only one modified file by the commit, as currently, most APR techniques focus on single file bugfixes.

Secondly, the modified file’s extension should match one of the extensions commonly used by the specified languages. Such precaution could be necessary so that it rules out cases when *e.g.* only a JSON file is modified during a bugfix.

Lastly, the code should not be a minified version of itself. JavaScript code is often minified to speed up webpage loading to improve website experience. However, for our purposes, we are only interested in code that is being developed by humans, for humans. And so, we rule out minified versions of code.

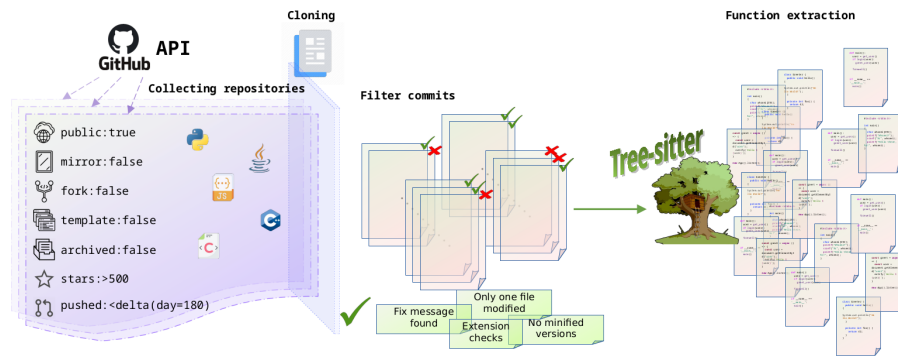


Figure 1: Overview of data mining procedure.

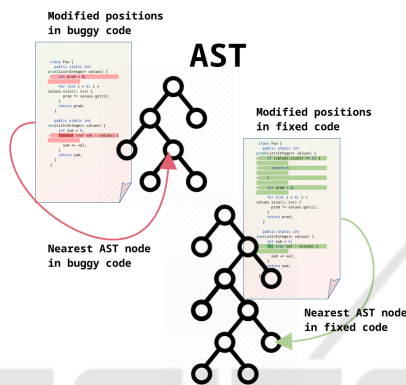


Figure 2: AST node position extraction.

Commits that abide by the above criteria are stored in JSON files. Most of the information is saved in the mentioned file, but it mainly contains the diff information, the buggy, and fixed version of the code, and the whole commit message.

Most importantly, the last step uses the Tree-sitter (Tree-sitter, 2023) library to extract specific code snippets from the source code. To the best of our knowledge, no other multi-lingual dataset uses the kind of post-processing we provide here. We generate an Abstract Syntax Tree (AST) representation of the buggy and fixed version of the source. We use the diff information (difflib, 2023) generated by comparing the buggy and the fixed code, extracting the positions of the applied changes. In other words, we map changes in the buggy file, to changes in the fixed version. This position mapping can then be used to find the nearest AST nodes relative to the applied code changes as depicted by Figure 2. By moving up in the node hierarchy *i.e.* querying the parent node by node, we check if a node’s type satisfies the condition of being of type function, method, function_declaration, *etc.* If the given condition is met, we store the code segment represented by that particular AST node.

```

class AwesomeCalculator {
    public static float divide(float a, float b) {
        if (a == 0) {
            return Double.NaN;
        }
        return a / b;
    }
}
    
```

Listing 1: Java code example.

Consider the buggy Java program in Listing 1 for a more concrete example. In this example, the fixed code should include the condition change from $a == 0$ to $b == 0$, and from the diff information we can find its position inside the tree constructed by Tree-sitter, in both the buggy and fixed version of the code, as shown by Listing 2. From there we can climb up the tree, search for a method declaration, and catch the encapsulating function. This could be done independently of the programming language so that this feature could easily be included in any source code miner software.

```

program [0, 0] - [8, 0]
  class_declaration [0, 0] - [7, 1]
    name: identifier [0, 6] - [0, 23]
    body: class_body [0, 24] - [7, 1]
      method_declaration [1, 4] - [6, 5]
        ...
        parameters: formal_parameters [1, 30] - [1, 48]
        ...
        body: block [1, 49] - [6, 5]
          if_statement [2, 8] - [4, 9]
            condition: condition [2, 11] - [2, 19]
            ...
            consequence: block [2, 20] - [4, 9]
            ...
            return_statement [5, 8] - [5, 21]
            ...
    
```

Listing 2: Tree generated by Tree-sitter.

The software could also be configured to catch the encapsulating class (or a given number of lines around a buggy segment, or any type of nodes), but for now, we chose to extract code segments on the function level, as it represents a relatively small, but complete code fragment. It is also notable, that this method is language agnostic thanks to the Tree-sitter (Tree-sitter, 2023) library. The method presented here, can also be found in our publicly available Github repository as an installable Python package. Outside of Tree-sitter, we do not have any special dependencies, and since it has bindings for many popular languages, our method can easily be implemented in those languages too, if one should have the need. Also, any other existing data mining software could make use of the *function-*, *class-*, or *line-by-line* environment extraction from source codes without needing to worry about the programming language, thus providing them with yet another useful feature.

5 DATASET

In this section, we will provide a detailed overview of the dataset. We will discuss metrics, possible use cases, and experiment designs that could be achieved using the dataset. We will also highlight some of the potential limitations and scenarios when the dataset cannot be used in its current form.

All in all, we collected more than 370000 buggy-fixed function pairs and their corresponding commit message. We also provide some metrics, which can be observed in Table 1. These metrics can help to give some insight into the size of separate parts of the dataset, which could provide helpful information for the user.

Table 1: Metrics describing the dataset.

Language	#Samples	#Chars (mean)	#Chars (med)	Size (MiB)
C	63928	1612.50	885.0	196.65
C++	69905	1503.97	828.0	200.60
Java	69917	1075.18	654.5	143.58
JavaScript	66453	1196.82	541.5	151.86
Python	101280	1168.78	693.5	225.94
Summary	371483	1311.45	720.5	918.63

#Samples is the number of buggy-fixed pairs extracted from commits under a specific language. #Chars mean, and median values provide some insight into the average source code length of the mined programs, while Size measures the UTF-8 encoded length of the source codes in mebibytes. The summary of sample and size information is a cumulative value of the separate samples, while both mean and median character values are averaged over the buggy and fixed codes.

5.1 Possible Use Cases

The CodegrainHouse dataset provides pre-processed buggy functions that can be used for several deep learning and generative tasks. Some of these tasks include:

1. *Bugfix Commit Generation*: The dataset can be used to train deep learning models to generate commit messages (Dong et al., 2022) for bugfixes. The pre-processed buggy, fixed, or both code segments can act as input to the DL model, so it can learn and generate relevant commit messages.

2. *Code Repair Prediction*: The dataset can predict the changes required to fix (Hu et al., 2022; Lu et al., 2021) the buggy functions. The pre-processed buggy functions serve as the input to the deep learning model, which can predict the appropriate code changes to repair the bugs.

3. *Code Completion and Suggestion*: The dataset can also be used for generative tasks, such as training deep learning models for code completion (Lu et al., 2021) and code suggestion. The fixed part of the dataset also consists of pre-processed code snippets, and thus could serve as input to such a generative model, which in turn could suggest appropriate code snippets to complete the code.

4. *Buggy Function Detection*: The dataset can train deep learning models for buggy function detection tasks (Alrashedy, 2023; Phan et al., 2021). The pre-processed buggy code sections can help the model learn the patterns in buggy code, making it a useful tool for identifying such functions. Such models could be used in a pre-commit hook, to mark these functions for revisal.

Example: for clarity, we show an example to provide more insight into the basic structure of the dataset. On Listing 3 the buggy code could be observed, while Listing 4 shows the fixed program. For the snippet that shows the fixed code, we put the bugfix commit message on top for simplicity, however, the dataset itself does not (necessarily) contain the commit message in the fixed source code in this format. We marked such messages as comments with two enclosures @ to avoid confusion.

The examples on Listings 3 and 4 shows us a simple bugfix, where an unnecessary `".join(...)"` statement is removed, returning the file contents as an array.

The dataset contains more complex and larger examples too, however for a short showcase of the bugs, we only included some that are short and relatively simple.

```
def readfile(filename):
    try:
        with open(filename, "r") as f:
            line = "".join(f.readlines())
    except Exception as e:
        log.error('Failed to read file "%s": %s' % (filename,
            ↪ e))
        return None
    return line
```

Listing 3: Buggy Python function.

```
# @@ firewall.functions.readfile: Return lines
↪ read from the file as an array
#
# This is needed for the dbug output in
↪ ipXtables.set_rules, ebtables.set_rules and
↪ ipset.restore. @@
def readfile(filename):
    i = 1
    try:
        with open(filename, "r") as f:
            return f.readlines()
    except Exception as e:
        log.error('Failed to read file "%s": %s' % (filename,
            ↪ e))
        return None
```

Listing 4: Fixed Python function.

5.2 Limitations

CodegrainHouse’s main usage is to train and validate deep-learning models. The dataset does not aim to be used with generate and validate (G&V) approaches (Martinez and Monperrus, 2016; He et al., 2023), where test cases should also be available for a given bug. Linking test cases to a given function is still a challenging task today, and the demand for existing tests would limit the number of repositories that could be crawled from the web, as not all projects include test cases for every possible bug. While this aspect of CodegrainHouse could be limiting, it also allows models to train on bugs not covered by test cases.

The dataset is relatively large in size, however, in itself, it may be insufficient to train large language models (Touvron et al., 2023; Brown et al., 2020) (LLM) like current Generative Pretrained Transformer (GPT) based models.

Also, the dataset currently only features 5 popular languages, which may pose a threat to data diversity, however, the scope of languages could be expanded in the future.

Furthermore, we tried to minimize the usage of Github API, so that only minimal interaction is required relieving much of the work from Github servers, however, this results in a higher space re-

quirement from the end user. While space requirements depend highly on the repository caching mechanism, the application still requires a large number of writes. Upon completing the cloning process, the user would probably have written more than 100 GiB of data while cloning the specified repositories. The dataset created is minimal in size compared to this - due to the extraction of bugfixing commits, and requested code snippets from the source code - but one should be aware of the immediate space requirements of the process.

6 CONCLUSION AND FUTURE WORK

In conclusion, we introduced a new dataset and a technique for extracting code fragments from source code on multiple granule levels using Tree-sitter (Tree-sitter, 2023). These levels can be used to include surrounding lines, the encapsulating function, or class. The provided dataset is publicly available as a compressed archive, and the methods used for code-snippet extraction are also publicly available in our GitHub repository.

In the future, we plan to release software (YAMI: Yet Another MIner), based on the techniques used in this paper, that could be easily used by researchers to mine their own datasets with their desired configurations. Future plans also include but are not limited to, using the currently mined dataset to see how current LLMs or other DL models perform in a multi-linguistic environment on APR tasks.

ACKNOWLEDGEMENTS

The research presented in this paper was supported in part by the European Union project RRF-2.3.1-21-2022-00004 within the framework of the Artificial Intelligence National Laboratory. The national project TKP2021-NVA-09 also supported this work. Project no TKP2021-NVA-09 has been implemented with the support provided by the Ministry of Culture and Innovation of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme.

REFERENCES

Alrashedy, K. (2023). Language Models are Better Bug Detector Through Code-Pair Classification.

- Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). Language Models Are Few-Shot Learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems, NIPS'20*, Red Hook, NY, USA. Curran Associates Inc.
- Bui, Q.-C., Scandariato, R., and Ferreyra, N. E. D. (2022). Vul4J: A Dataset of Reproducible Java Vulnerabilities Geared Towards the Study of Program Repair Techniques. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, pages 464–468.
- Chen, Z., Komrusch, S. J., Tufano, M., Pouchet, L. N., Poshyanyk, D., and Monperrus, M. (2019). SEQUENCER: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Transactions on Software Engineering*, (01):1–1.
- Csuvik, V. and Vidács, L. (2022). FixJS: A Dataset of Bug-Fixing JavaScript Commits. In *Proceedings of the 19th International Conference on Mining Software Repositories, MSR '22*, page 712–716, New York, NY, USA. Association for Computing Machinery.
- difflib (2023). difflib. <https://docs.python.org/3.11/library/difflib.html>.
- Dinella, E., Dai, H., Brain, G., Li, Z., Naik, M., Song, L., Tech, G., and Wang, K. (2020). Hoppity: Learning Graph Transformations To Detect and Fix Bugs in Programs. Technical report.
- Dong, J., Lou, Y., Zhu, Q., Sun, Z., Li, Z., Zhang, W., and Hao, D. (2022). FIRA: μ fi/ U_{μ} ne-Grained G_{μ} ra/ U_{μ} ph-Based Code Change Representation for Automated Commit Message Generation. *ICSE '22*, page 970–981, New York, NY, USA. Association for Computing Machinery.
- Drain, D., Wu, C., Svyatkovskiy, A., and Sundaresan, N. (2021). Generating bug-fixes using pretrained transformers. *MAPS 2021 - Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming, co-located with PLDI 2021*, pages 1–8.
- GitHub (2023a). GitHub. <https://github.com/>.
- GitHub (2023b). Octoverse: The state of open source and rise of AI in 2023. <https://octoverse.github.com>.
- GitHub REST API (2023a). GitHub REST API Official Website. <https://docs.github.com/en/rest>.
- GitHub REST API (2023b). Rate limits of GitHub's REST API. <https://docs.github.com/en/rest/overview/rate-limits-for-the-rest-api?apiVersion=2022-11-28>.
- GitPython (2023). GitPython. <https://github.com/gitpython-developers/GitPython>.
- Gyimesi, P., Vancsics, B., Stocco, A., Mazinanian, D., Beszédes, A., Ferent, R., and Mesbah, A. (2019). BugsJS: a Benchmark of JavaScript Bugs. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 90–101.
- He, Y., Chen, Z., and Le Goues, C. (2023). PreciseBug-Collector: Extensible, Executable and Precise Bug-Fix Collection: Solution for Challenge 8: Automating Precise Data Collection for Code Snippets with Bugs, Fixes, Locations, and Types. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1899–1910.
- Hu, Y., Shi, X., Zhou, Q., and Pike, L. (2022). Fix Bugs with Transformer through a Neural-Symbolic Edit Grammar. In *Deep Learning for Code Workshop*.
- Jiang, N., Lutellier, T., and Tan, L. (2021). CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. pages 1161–1173.
- Just, R., Jalali, D., and Ernst, M. D. (2014). Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *2014 International Symposium on Software Testing and Analysis, ISSTA 2014 - Proceedings*, pages 437–440. Association for Computing Machinery, Inc.
- Karampatsis, R.-M. and Sutton, C. (2020). How Often Do Single-Statement Bugs Occur? The ManySStuBs4J Dataset. In *Proceedings of the 17th International Conference on Mining Software Repositories, MSR '20*, page 573–577, New York, NY, USA. Association for Computing Machinery.
- Kim, M., Kim, Y., Jeong, H., Heo, J., Kim, S., Chung, H., and Lee, E. (2022). An Empirical Study of Deep Transfer Learning-Based Program Repair for Kotlin Projects. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ES-EC/FSE 2022*, page 1441–1452, New York, NY, USA. Association for Computing Machinery.
- Le Goues, C., Holtschulte, N., Smith, E. K., Brun, Y., Devanbu, P., Forrest, S., and Weimer, W. (2015). The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256.
- Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, D., Tang, D., Li, G., Zhou, L., Shou, L., Zhou, L., Tufano, M., Gong, M., Zhou, M., Duan, N., Sundaresan, N., Deng, S. K., Fu, S., and Liu, S. (2021). CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *undefined*.
- Lutellier, T., Pang, L., Pham, V. H., Wei, M., and Tan, L. (2019). ENCORE: Ensemble Learning using Convolution Neural Machine Translation for Automatic Program Repair.
- Lutellier, T., Pham, H. V., Pang, L., Li, Y., Wei, M., and Tan, L. (2020). CoCoNuT: Combining context-aware neural translation models using ensemble for program repair. *ISSTA 2020 - Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 20:101–114.
- Martinez, M. and Monperrus, M. (2016). ASTOR: A program repair library for Java (Demo). *ISSTA 2016 - Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 441–444.
- Monperrus, M. (2020). The Living Review on Automated Program Repair. Technical report.

- Phan, L., Tran, H., Le, D., Nguyen, H., Annibal, J., Peltekian, A., and Ye, Y. (2021). CoTexT: Multi-task Learning with Code-Text Transformer. pages 40–47.
- Ponta, S. E., Plate, H., Sabetta, A., Bezzi, M., and Dangremont, C. (2019). A Manually-Curated Dataset of Fixes to Vulnerabilities of Open-Source Software. In *Proceedings of the 16th International Conference on Mining Software Repositories*.
- Saavedra, N., Silva, A., and Monperrus, M. (2023). Github-actions: Building reproducible bug-fix benchmarks with github actions.
- Tan, S. H., Yi, J., Yulis, Mechtaev, S., and Roychoudhury, A. (2017). Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 180–182.
- Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., and Lample, G. (2023). LLaMA: Open and Efficient Foundation Language Models.
- Tree-sitter (2023). Tree-sitter. <https://tree-sitter.github.io/tree-sitter/>.
- Tufano, M., Pantiuchina, J., Watson, C., Bavota, G., and Poshyvanyk, D. (2019a). On learning meaningful code changes via neural machine translation. In *Proceedings of the 41st International Conference on Software Engineering, ICSE '19*, page 25–36. IEEE Press.
- Tufano, M., Watson, C., Bavota, G., Penta, M. D., White, M., and Poshyvanyk, D. (2019b). An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. *ACM Trans. Softw. Eng. Methodol.*, 28(4).
- Widyasari, R., Sim, S. Q., Lok, C., Qi, H., Phan, J., Tay, Q., Tan, C., Wee, F., Tan, J. E., Yieh, Y., Goh, B., Thung, F., Kang, H. J., Hoang, T., Lo, D., and Ouh, E. L. (2020). BugsInPy: A Database of Existing Bugs in Python Programs to Enable Controlled Testing and Debugging Studies. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2020*, page 1556–1560, New York, NY, USA. Association for Computing Machinery.
- Xu, S., Yao, Y., Xu, F., Gu, T., Tong, H., and Lu, J. (2019). Commit message generation for source code changes. In *IJCAI*.
- Ye, H., Martinez, M., Durieux, T., and Monperrus, M. (2019). A Comprehensive Study of Automatic Program Repair on the QuixBugs Benchmark. *IBF 2019 - 2019 IEEE 1st International Workshop on Intelligent Bug Fixing*, pages 1–10.
- Yi, L., Wang, S., and Nguyen, T. N. (2020). Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings - International Conference on Software Engineering*, pages 602–614. IEEE Computer Society.