PyTraceBugs: A Large Python Code Dataset for Supervised Machine Learning in Software Defect Prediction

Elena N. Akimova*†, Alexander Yu. Bersenev*†, Artem A. Deikov*†, Konstantin S. Kobylkin*†,
Anton V. Konygin*, Ilya P. Mezentsev*†, Vladimir E. Misilov*†
* Krasovskii Institute of Mathematics and Mechanics, UB RAS, S. Kovalevskaya Street 16, 620108 Ekaterinburg, Russia
† Ural Federal University, Mira Street 19, 620002 Ekaterinburg, Russia
{aen, kobylkin, konygin}@imm.uran.ru, {alexander.bersenev, deykov.artem, ilya.mezentsev, v.e.misilov}@urfu.ru

***Abstract*—Contemporary software engineering tools employ deep learning methods to identify bugs and defects in source code. Being data-hungry, supervised deep neural network models require large labeled datasets for their robust and accurate training. In distinction to, say, Java, there is lack of such datasets for Python. Most of the known datasets containing the labeled Python source code are of relatively small size. Those datasets are suitable for testing built deep learning models, but not for their training. Therefore, larger labeled datasets have to be created based on some well-received algorithmic principles to select relevant source code from the available public codebases.**

In this work, a large dataset of the labeled Python source code is created named PyTraceBugs. It is intended for training, validating, and evaluating large deep learning models to identify a special class of low-level bugs in source code snippets manifested by throwing error exceptions, reported in standard traceback messages. Here, a code snippet is assumed to be either a function or a method implementation. The dataset contains 5.7 million correct source code snippets and 24 thousands buggy snippets from the Github public repositories. Most represented bugs are: absence of attribute, empty object, index out of range, and text encoding/decoding errors.

The dataset is split into training, validation and test samples. Confidence in labeling of the snippets into buggy and correct is about 85% according to our estimates. Labeling of the snippets in the test sample is additionally manually validated to be almost 100% confident.

To demonstrate advantages of our dataset, it is used to train a binary classification model for distinguishing the buggy and correct source code. This model employs the pretrained BERT-like contextual embeddings. Its performances are as follows: precision on the test set is 96% for the buggy source code and 61% for the correct source code whereas recall is 34% and 99% respectively. The model performance is also estimated on the known BugsInPy dataset: here, it reports approximately 14% of buggy snippets.

***Index Terms*—defect prediction, bug dataset, data mining**

I. Introduction

Identifying deficient source code is a longstanding problem in the software engineering. Bugs in software not only increase development time but can also lead to unexpected program failures, possibly incurring significant money losses. Numerous software analysis tools are created to assist developers in finding and fixing bugs. These tools can roughly be divided into two groups: static and dynamic analyzers. Static tools try to identify bugs in source code without compiling it whereas non-static ones also employ the byte code data. In this work, a bug prediction problem is considered in the Python programs based on their static source code analysis. Python has become a language of choice for many developers working in a variety of domains including the web development, data science, and machine learning. It has its own specifics such as dynamic typing and overwhelming object paradigm in distinction to the other languages such as Java and C#, which hamper finding bugs. Moreover, the Python interpreter and existing static analysis tools do not provide any thorough static checks for source code of snippets, postponing revealing of bugs to runtime stage. This leads to the need of debugging programs, which might be costly.

Modern static analysis tools and software use a variety of methods including type inference, safe refactoring, enforcing coding standards, as well as monitoring specific numeric metrics computed for source code. Recent advances in machine learning, especially, in deep learning, provide not only a step forward towards improving quality of those methods, but, also, propose a powerful alternative to them. For example, some of the existing deep learning models trained on source code [1] employ mining relations between source code entities to be utilized in bug prediction, localization, and repair. This results in significant improvement over the state-of-the art results. In the present work, this mainstream line of research is followed to apply deep learning to source code aimed to propose more advanced static analysis tools.

Deep learning models, such as Transformers [2] and graph neural networks [3], are extremely data-hungry as they have hundreds of millions of parameters to be estimated even during fine-tuning. There are two ways to collect a large amount of relevant data for training and evaluating these models. The first one is to artificially generate the training and validation samples based on analysis of real source code changes introduced during bugfixing process. Under this approach, correct source code is corrupted according to some principles of transforming it to buggy, learned from real bugfix changes, made by developers. The corrupted source code gives examples of buggy snippets whereas uncorrupted one forms examples of the correct source code in training and validation samples. Here, the corresponding test dataset is usually a small sized manually curated dataset of real bugs collected from existing bugfixes stored in version control systems of real projects.

The main advantage of this approach consists in the fact that an arbitrarily large dataset can be created for training models. Its drawback is that such a dataset does not reflect actual proportions of bug types in the source code. Moreover, it is confined to some prescribed patterns of corrupting source code to be used to convert correct code into buggy. This may result in a certain bias in performance of the bug detectors trained on such datasets.

The second way consists in using real data for both training and evaluating deep learning models. Within this approach, both training and validation samples are composed of real examples of buggy and correct source code snippets, automatically collected from available codebases. Here, the data collection process is guided by some well-understood principles to select relevant source code. To increase confidence of labeling of snippets in the corresponding test sample, it is composed of manually selected snippets analogously to the first approach.

Advantage of this approach is that the model learns from real bugs and works in more or less the same conditions during its training and prediction. As for its drawbacks, the collected data contain some amount of noise, *e.g.*, either false positives in identifying buggy snippets or false negatives for the correct source code. Here, the proportion must be estimated of such noisy labeling.

In our work, the latter approach is adopted due to the following reasons. The first approach fits best when patterns to corrupt correct source code snippets are more or less easy to understand and learn. This is not the case for other types of simple bugs.

In the second approach, it is left for models to learn typical patterns of textual difference between buggy and correct snippets from real data. This comes at cost because changes introduced into a buggy snippet are not necessarily confined to fixing bugs. They might include refactoring or other changes improving the code quality.

In this work, a PyTraceBugs dataset is created both for training and evaluating of deep learning models for bug prediction at the granularity of Python functions and methods. More specifically, it is aimed to build predictive models for classical bug prediction problem setting. In this setting, there are two classes of snippets, which either contain or do not contain bugs. The dataset is split into training, validation, and test samples where the test sample is guaranteed to have almost 100% confident labeling of its snippets. Moreover, our estimate is 85% of confidence in labeling of the training and validation samples.

The rest of this paper is structured as follows. Section II presents some related work. Section III overviews the dataset and describes its content. Section IV assesses quality of the dataset. Section V presents statistics of the code that is included in the dataset. Section VI describes a bug prediction model trained on our dataset and presents its performance. Section VII describes in-depth the dataset collection procedures including parallelization. Finally, we conclude our work in Section VIII.

II. Related Work

In related work, several datasets, which contain buggy source code from a variety of programming languages, are described. Some of these datasets are intended for developing and evaluating automatic testing and debugging tools. They emphasize the features that are valuable for this problem, such as reproducibility of a bug and isolation of a fix. The former means that each entity should be accompanied by a test-suite that gives consistent results (failing on the buggy version of code and passing on the fixed one). The latter means that the buggy and fixed code should differ only by a bug fix, which does not include irrelevant changes, such as refactoring and adding features. Most popular datasets of this type for Java and C languages are ManyBugs [4] (185 C bugs) Defects4J [5] (357 Java bugs), Bugs.jar [6] (1,158 Java bugs). The BugsInPy dataset [7] contains 493 bugs collected from Python projects.

These datasets are usually manually curated, which ensures their high quality. However, this type of datasets is not suitable for training machine learning models for defect-related problems due to their relatively small size, moreover, reproducibility and isolation are less relevant for static code analysis.

Larger datasets are usually automatically created. The BugSwarm dataset [8] contains reproducible bugs from open source projects, which are mined from the GitHub platform, using continuous integration services. It extracts bug-fix pairs by detecting the code, which fails to build and its subsequent version that passes. It contains 1,940 pairs of the Java files and 1,292 pairs of the Python files.

The common method for automatic collection of large labeled code datasets is to extract data from versioning and bug reporting systems. The main idea is to link reported problems, bugs, and issues with the subsequent code changes to isolate the bugfixing changes and then extract the corresponding buggy and fixed version of the source code. Most popular algorithm is SZZ [9]–[11]. It consists in identifying the bug fixing commit and using the commit message to find the matching issue. However, recent studies [12], [13] show that this approach is prone to mislabeling. Many files found by the basic SZZ algorithm are not the “pure” bug fixes. The mistakenly labeled files may contain changes in the test code, comments, or code refactoring. In our work, special measures are taken to avoid such problems and ensure validity of labeling.

The collected data usually serve multiple purposes. The first one is the automated program repair. It consists in generating a change that fixes the buggy code (see survey [14]). Constructing models for this problem implies analyzing bugs and changes in the dataset to extract and reproduce the patterns that are used to repair the code.

There are several datasets constructed for this task. The CodRep dataset [15] consists of five parts containing one-line bugfixes from open source Java projects. Its total size is 58,069 commits. The work [16] uses this artifact for program repair.

The dataset from [17] consists of bugfix pairs of Java code mined from the GitHub. Its authors identified all commits with messages like “fix”, “issue” or “bug”. For each of such commits, the pair of buggy (precommit) and fixed (postcommit) code fragments is extracted. The resulting dataset contains 800 thousands code pairs.

The ManySStuBs4J dataset [18] is constructed in the same manner. It consists of two parts: 25,539 single-statement (one-line) bugfix pairs from 100 popular open-source Java projects and a larger set of 153,652 pairs from 1,000 projects. The authors extracted only those commits, for which both buggy and fixed code passes compilation. So, this dataset is confined to simple bugs, which are difficult to spot manually. The non-bugfixing changes, such as renaming variables or classes, are excluded. The resulting bugfixes form 16 patterns such as variable misuse, wrong function name, wrong operator, *etc.*

The second application is the software defect prediction. Here, the goal is to identify a given code excerpt (file, class, function, or method implementation) as defective or correct (see recent survey in [19]). The defect prediction problem can be posed in different settings. One of the approaches is to consider it as a binary classification problem. Below, are some of the datasets intended to be used in training predictive models for this setting.

The GHPR dataset [20] consists of 3,026 pairs of defective and fixed source code files from 307 projects. They are extracted from the GitHub by finding pull requests labeled as fixes. Then, the resulting code pairs are used to train a graph-based deep learning model for defect prediction.

The BugHunter dataset [21] contains 159 thousands Java bugs with precomputed metrics for three granularity levels (file/class/method). It is created from GitHub projects by analyzing the closed and open bug reports containing references to the bugfixing commits. The authors use the computed metrics of code to train classifiers for defect prediction.

In the works above, buggy excerpts are distinguished from their instant fixes, *i.e.*, the parts of the bugfix pairs are discriminated from each other. In our work, a different setting is proposed, which does not treat fixed versions of buggy code as correct. Namely, it is assumed that a snippet is correct if it is stable, *i.e.*, has not been changed for a significant time period up the current state of software. This definition of correct source code is more preferable, as guarantees of correctness can not be provided for fix snippets. It is due to the fact that changes can be further introduced into those snippets because of many reasons including fixing other bugs. This idea lies at the core of collecting our dataset.

III. The PyTraceBugs dataset

In this work, a large dataset of Python source code is presented called the PyTraceBugs dataset. This dataset is intended for both training and evaluating complex (possibly, neural) predictive models for software bug prediction. It is formed by excerpts of source code of functions and methods from the selected top rated GitHub repositories. It contains examples from both classes of the correct and buggy source code.

A range of possible bugs conveyed by the dataset is somewhat restrictive in a sense that it only contains examples of confirmed bugs. These are the bugs, which manifest themselves in the form of raising an error (exception) message, which is called the traceback error message in Python. This kind of restriction serves two goals. First, this sort of bugs can be considered as low-level, *i.e.*, snippet level. This is the kind of bugs we are aimed to find. Second, it could simplify the bug localization to some extent as many bugs tend to be near traceback paths in the graph of function calls.

Totally, the dataset contains 24 thousands examples of buggy and 5.7 million correct source code snippets from 630 and 10,642 repositories, respectively. Repositories for the buggy source code employ the standartized GitHub system for handling issues and fixing bugs in their codebase.

In Table I top 20 most present GitHub repositories are listed accounting approximately 40% of snippets of buggy code.

TABLE I
Top repositories per number of buggy snippets

Repository name	**Percentage of snippets (%)**
saltstack/salt	7.5%
ansible/ansible	5.9%
Tribler/tribler	3%
pandas-dev/pandas	3%
mars-project/mars	2.6%
numba/numba	2.5%
spyder-ide/spyder	2.3%
Cog-Creators/Red-DiscordBot	1.8%
pymedusa/Medusa	1.5%
python/mypy	1.4%
conda/conda	1.3%
log2timeline/plaso	1.3%
ray-project/ray	1.1%
scikit-learn/scikit-learn	1%
freqtrade/freqtrade	1%
pytroll/satpy	1%
iterative/dvc	0.9%
modin-project/modin	0.9%
sphinx-doc/sphinx	0.9%
scipy/scipy	0.8%
Others	58.3%

The corresponding distribution of snippets for the correct source code is presented in Table II. Here, top 25 repositories account about 15% of snippets. Approximately 16% of the repositories of the buggy source code are also present in the correct source code, though, with smaller percentages of snippets.

A. Content of the dataset

The dataset consists of two parts:

- automatically collected and filtered source code, containing examples of buggy and correct snippets;
- a small sample of this automatically collected data, which is subjected to additional filtering and manual validation by two Python experts.

Here, the programmatically collected source code is split into training and validation samples, whereas its manually selected sample is used as a test sample.

The basic principles of automatic selection of examples of correct and deficient code snippets from the GitHub repositories for both training and validation samples are given below. With some specific modifications, they are used in works [7], [21] as a first step to collect datasets of source code for other programming languages and other research purposes, *e.g.*, for datasets aimed to improve automatic test generation.

Namely, snippets of the correct source code are chosen from stable code of the GitHub repositories under a simple assumption, which (omitting the details) amounts to the following one: a snippet is more probably correct if it has not been changed over many commits up to the latest state of the repository folder it resides in.

Snippets of source code with bugs are collected from codebase of the top GitHub repositories with issues pages. More specifically, they are selected from those repositories bugfix commits and pull requests, which are directly related to handling issues marked by bug labels, *e.g.*, named as "bug", "type:bug" or have any other similar label.

Besides, only those bugs are considered, which manifest themselves by raising an error exception. Accordingly, the repositories issues are selected to contain full error traceback reports on their web pages. These issues report a program crash, which developers consider as a bug and fix it. Table III contains error exception types with maximal occurence in the dataset.

The most frequent error messages are the attribute absence, empty object related errors, and index out of bounds error.

IV. Quality of the Dataset

A. Labeling confidence

To enforce high confidence of labeling of snippets of the test sample, a manual validation process is conducted to select relevant snippets from the programmatically collected source code. Below, ideas are outlined to select examples of the buggy code:

- a bug reported on the web page of the corresponding issue is simple and easy to understand (*e.g.*, its actual fix appears near the location where the program crashes, raising an error exception)
- the reported bug is not dependency, compatibility, or the regression bug;
- a fix introduced into a buggy snippet is also simple, *e.g.*, is confined to one line;
- changes (introduced into the snippet) should not be bound to refactoring, *i.e.*, they must be changes fixing the reported bug.

TABLE II
Top repositories per number of correct snippets

Repository name	**Percentage of snippets (%)**
CiscoDevNet/ydk-py	4.5%
Azure/azure-sdk-for-python	1.6%
oracle/oci-python-sdk	0.7%
sagemath/sage	0.7%
cloudera/hue	0.7%
tytusdb/tytus	0.6%
aliyun/aliyun-openapi-python-sdk	0.5%
home-assistant/core	0.5%
cctbx/cctbx_project	0.5%
tencentyun/scf-demo-repo	0.5%
Azure/azure-cli-extensions	0.5%
sympy/sympy	0.4%
dimagi/commcare-hq	0.4%
kovidgoyal/calibre	0.4%
docusign/docusign-python-client	0.3%
saltstack/salt	0.3%
leo-editor/leo-editor	0.3%
tribe29/checkmk	0.3%
XX-net/XX-Net	0.3%
Toontown-Open-Source-Initiative/ Toontown-School-House	0.3%
AppScale/gts	0.3%
anhstudios/swganh	0.3%
SteveDoyle2/pyNastran	0.3%
dnanexus/parliament2	0.3%
openhatch/oh-mainline	0.2%
Others	84.8%

TABLE III
Most common error types in the dataset

Error type	**Percentage of snippets (%)**
AttributeError	16.6%
TypeError	15.9%
ValueError	10.2%
KeyError	8.2%
RuntimeError	5.5%
IndexError	5.3%
Others	38.3%

These tough restrictions confined 1 out of approximately 15–20 snippets during the validation process. The process consisted in reviewing and selecting the buggy snippets among a random sample of several hundreds of the automatically collected entries.

To filter out the correct snippets with highly confident labeling to be included into the test sample, the following automatic selection principle is used. Namely, snippets of stable source code are subjected to an additional restriction that selects only those snippets, which have many incoming calls from other snippets with many incoming calls. Namely, a graph of calls is computed for snippets of each GitHub repository chosen to be a source of stable code for the dataset:

snippet 1 is adjacent to snippet 2, if there is a call of snippet 2 in the implementation of snippet 1. A snippet of stable code is chosen to be included in the test sample, if there are at least 3 snippets with incoming degrees above 3, which are adjacent to that snippet. In our opinion, this selection criterion must increase visibility of bugs in the source code of selected snippets, and thus, provides additional guarantees that bugs in those snippets are more likely to be already revealed and fixed.

The buggy snippets from training and validation samples have more or less confident labeling due to the following specifics of the source code selection from the GitHub bugfix commits and pull requests:

1) buggy snippets are being a part of the GitHub bugfixes directly related to issues reporting a program crash in the form of raising an error exception;
2) being selected from the top rated GitHub repositories issues pages, these issues have bug labels; this increases the confidence that the reported problems with repositories source code are bound to be bugs;
3) only highly relevant bugfix commits and pull requests are considered: *i.e.*, they should contain a direct reference to a buggy labeled issue in their message or title and, if it occurs in message, the prescribed tokens should precede this reference *e.g.*, “closes”, “fixes”, *etc.*;
4) bugfix commits and pull requests are additionally restricted to handle at most two issues.

To directly estimate quality of the dataset, a percentage is estimated of buggy snippets, for which the corresponding fixes from the bugfix commits and pull requests contain changes, being confined to refactoring. Such changes are not directly related to bugs. Therefore, the corresponding snippets should be excluded from the collected data as being not correctly labeled. To obtain a lower bound on the percentage of the refactoring changes present in the training and validation samples, the rate is estimated for changes from the corresponding bugfix commits, which are bound to docstrings and comments. Namely, it equals to 2.6%. Besides, during the manual validation process to form the test sample, our experts observed approximately 10–15% of refactoring changes.

To guarantee confidence of labeling of correct source code from training and validation samples, only highly stable source code is selected from the top rated GitHub repositories. More specifically, a count of commits is computed for each snippet of source code, which is relative to the directory where this snippet is located. It counts the number of the commits, which contain changes for at least one *.py* file in this specific directory, but make no changes in that snippet. Only the snippets are selected to be included into training and validation samples, whose corresponding count is above 100.

B. Ability of the cross-project and cross-domain prediction

The repositories of the training and test samples are made non-overlapping. This was done to avoid possible data leaks and provide non-biased performance estimation for models trained on this dataset. This allows cross-project predictions.

To explore difference of domains across the training, validation, and test samples, topics are also collected for each repository present in the dataset. In the GitHub repositories topics are represented by lists of keywords in the main repositories pages.

For the buggy source code, 562 distinct repositories are present in the training sample, whereas 68 repositories are present both in the validation and test samples. They have 2,345, 424, and 424 distinct topics, respectively. Here, the validation and test samples share the same topics, whereas the test sample has 228 topics not present in the training sample.

For the correct source code, 1,571, 10,536, and 94 repositories are present in the training, validation and test samples with 3,456, 14,066, and 269 distinct topics, respectively. There are 89 topics in the test sample not present in the validation sample. Besides, 162 topics of the test sample are not present in the training sample.

This, to some extent, provides an evidence that the dataset allows cross-domain predictions.

V. Summary statistics for the dataset

Training and validation samples include 14,089 and 9,457 snippets of the buggy code, as well as, 351,338 and 5,340,000 snippets of the correct source code respectively. The test sample contains 170 snippets of the correct source code and 161 snippets of the buggy code.

Table IV contains statistics for distribution of the buggy code with respect to two its characteristics. The first one is its length in symbols including docstrings and comments. The second characteristic is the well known cyclomatic complexity, which is a special measure of structural complexity expressed in the form of number of logical paths in source code. For example, if a snippet does not contain branching statements, *e.g.*, in if-else statements, the cyclomatic complexity is equal to 1; when source code contains a single if-else block statement, its cyclomatic complexity is equal to 2.

TABLE IV
Statistics of the buggy code

Statistic name	Code length	Cyclomatic complexity
Mean	1,875.9	7.0
Median	934.5	3.0
Standard error	3,766.9	12.1
25%	426.0	1.0
75%	2,010.0	8.0
90%	4,081.3	16.0
Min	19	0
Max	185,063	199

Statistics are given in Table V for distribution of stable code with respect to its length and cyclomatic complexity.

Thus, stable source code is generally shorter and simpler than buggy one.

Topics of repositories with the buggy code are presented in Table VI. Here, the snippets can have multiple topics. Many of the topics are devoted to infrastructure and cloud computing,

TABLE V
STATISTICS OF THE STABLE CODE

Statistic name	Code length	Cyclomatic complexity
Mean	704.87	2.16
Median	322.0	1.0
Standard error	4,089.28	5.66
25%	151.0	0.0
75%	734.0	2.0
90%	1,596.0	6.0
Min	13	0
Max	6,804,973	2,640

as well as, to the data science. Most popular topics for the correct source code are given in Table VII. A large bulk of topics is devoted to the data science and IOS development.

TABLE VI
TOPICS OF THE BUGGY CODE

Topic name	Percentage of snippets (%)
python	69.6%
hacktoberfest	18.6%
machine-learning	12.1%
cloud	8.8%
pandas	8.5%
iot	7.7%
infrastructure-as-code	7.6%
cloud-management	7.6%
event-management	7.6%
infrastructure-as-a-code	7.5%
zeromq	7.5%
infrastructure-management	7.5%
cloud-provisioning	7.5%
cloud-providers	7.5%
infrasructure	7.5%
remote-execution	7.5%
infrastructure-automation	7.5%
edge	7.5%
event-stream	7.5%
numpy	7.5%
pytorch	7.3%
numpy	6.7%
data-science	6.4%
ansible	6.2%
deep-learning	5.1%
data-analysis	4.8%

VI. TRAINING AND EVALUATING BUG PREDICTION MODELS ON THE DATASET

Another way to estimate quality of the dataset consists in building predictive models using its data. Namely, a binary classification problem is considered, in which a given source code snippet is classified into either buggy or correct. Here, a probability is estimated for the snippet to contain bugs. The standard logloss is used as a loss criterion, whereas precision and recall are performance metrics for both validation and test samples.

In order to apply machine learning models for the considered binary classification problem, feature engineering is first conducted for training, validation, and test samples. In this work, special Transformer-based embeddings are employed for this purpose. More specifically, the multi-language pretrained CodeBERT model [22][1] is applied. It uses exactly the same model architecture as the RoBERTa-base, which includes 12 layers. Each layer has 12 self-attention heads. Size of each head is 64. The hidden dimension is 768 and inner hidden size of the (pointwise) feed-forward layer is 3,072. Total number of model parameters is 125M.

First, each source code snippet is tokenized using the pretrained CodeBERT tokenizer (which is mostly a standard RoBERTa one). It extracts Python keywords, as well as, other words and terms as tokens. The hidden state of the last layer of the CodeBERT model is computed for each snippet token. The snippets are fed as input for the model without prelminarily removing docstrings and comments. Maximum model input length is 512 tokens. Longer source code fragments are truncated to first 512 tokens.

TABLE VII
TOPICS OF THE CORRECT CODE

Topic name	Percentage of snippets (%)
python	39.6%
hacktoberfest	7.1%
machine-learning	5.2%
catalyst	4.5%
yang	4.5%
ydk	4.5%
ios-xe	4.5%
nx-os	4.5%
ios-xr	4.5%
deep-learning	3.9%
python3	2.8%
pytorch	2.5%
science	2.5%
django	2.4%
tensorflow	2.2%
azure	2%
linux	2%
data-science	1.9%

The hidden state is represented by a vector of 768 numeric features. Hidden states of tokens are averaged to obtain a final embedding for the snippet. Such an embedding is employed as it is a good representation of the snippet tokens and does not lead to information loss. No further fine-tuning is conducted for the CodeBERT.

A LightGBM gradient boosting model is trained on the training sample. Parameters of the classifier are mostly set to default except the number of iterations and weights for classes. Number of iterations is chosen using validation and

[1]https://github.com/microsoft/CodeBERT

set to 2,000. To treat imbalance in the training sample, weights are introduced: for positive (buggy) snippets the corresponding weight is equal to the ratio of the counts of correct and buggy code snippets.

Table VIII presents the experimental results for the test sample of 330 snippets. Here, precision is 0.96 in the positive class, which means that labeling of the snippets classified as buggy is 96% confident. Moreover, among all snippets with bugs, 34% are labeled as buggy.

Besides, to implicitly check quality of our dataset, the model performance is also estimated on the known manually curated BugsInPy dataset, which is collected under the principles, similar to those, used to collect our dataset. Here, the model correctly selects about 14% of all buggy snippets. This could be due to the fact that it has been trained to identify only a special class of bugs, manifested by throwing exceptions, described in traceback reports. In the work [7], describing the BugsInPy dataset, no clear distinction is given of whether the present bugs cause programs throw exceptions.

TABLE VIII
RESULTS OF THE PREDICTION EXPERIMENTS

	Precision	Recall	F_1-measure
correct	0.61	0.99	0.76
buggy	0.96	0.34	0.5

Two variations of this experiment are also conducted. Namely, the first variation is done using much larger random sample of 1 millon snippets from the correct source code for the training sample. The second experiment involves another approach to selecting correct source code to be included into the test sample: 90 snippets of the correct code are randomly chosen to be included into the test sample, such that the corresponding repositories, containing them, are those, which also contain snippets of buggy source code of the test sample; the rest 80 snippets of the correct source code are chosen randomly. The trained models performance results for both variations are similar to those for the experiment above.

The experiment is also repeated on short source code snippets due to the CodeBERT model restriction for snippets to have at most 512 tokens. All snippets longer than 512 tokens are removed from the training, validation, and test samples. For the correct source code, 295,094 snippets are selected out of 351,338 of the training sample, whereas 144 out of 170 fragments are selected for the test sample. For the buggy source code, 8,537 snippets are chosen out of 14,089 of the training sample, and 83 out of 161 fragments are remained in the test sample.

Table IX contains the experimental results for the selected subsample of the test sample. Thus, exclusion of longer snippets disturbs performance metrics up to 0.1.

An experiment is also conducted where the software metrics are employed. Here, the well known raw, Halstead, and complexity metrics are computed for the training, validation and test samples using the known Radon library[2]. All those 22 metrics more or less reflect source code complexity from different points of view. Here, raw complexity amounts to the number of lines of source code, structural complexity gives the number of operators and operands whereas logical complexity reports the number of different logical paths in the source code. Computed metrics are combined with the CodeBERT features to produce a predictive LightGBM model.

TABLE IX
RESULTS FOR THE SHORT SNIPPETS

	Precision	Recall	F_1-measure
correct	0.7	1.0	0.82
buggy	1.0	0.24	0.39

Results on the test sample are presented in Table X. Thus, adding software metrics to the CodeBERT features increases F_1-measure for the buggy class from 0.5 to 0.7.

TABLE X
RESULTS FOR THE CODEBERT AND METRICS

	Precision	Recall	F_1-measure
correct	0.78	0.69	0.73
buggy	0.71	0.8	0.75

A. *Importance of the CodeBERT features for defect prediction*

An experiment is conducted to estimate predictive importance of the CodeBERT embeddings. Here, the importance is measured using somewhat non-standard method: it estimates a degree, to which coordinates of the CodeBERT embedding vectors change during the bugfixing process. Namely, as the buggy snippets in the dataset come from the GitHub bugfixing commits and pull requests, each buggy snippet has a pair being a snippet, which represents the most earlier (instant) fix of bugs in the buggy snippet.

Thus, to measure importance of the CodeBERT embeddings, one can estimate magnitudes of differences of coordinates of the CodeBERT embeddings computed both for the buggy snippets and their corresponding paired fix snippets. In other words, embeddings of two states of the same snippet are compared right before and after bugfixing changes are introduced in that buggy snippet.

The adopted method to measure importances is different from the common one where the importances are measured directly with respect to a given predictive model. In our opinion, the former method better reflects sensitivity of embeddings to the bugfixing process as the fix source code snippet contains changes made to fix only a small number or even a single bug in a short time period. In distinction to this, there can be a long path of bugfixes between a buggy snippet and its stable version.

[2]https://pypi.org/project/radon/

To explore distributions of differences between embeddings of the source code before and after bug fixing, all programmatically collected data are used (approximately 24 thousands bugfix pairs). For each such bugfix pair of snippets, their 768-dimensional CodeBERT embeddings and the corresponding differences are computed. Let them be called "bug minus fix differences". For each component of bug minus fix differences the Wilcoxon signed-rank test is applied and its p-value is computed. It tests whether distribution of the bug minus fix differences is symmetric about zero. Then, a multiple testing correction (Benjamini–Hochberg) procedure is applied to the list of 768 computed p-values. It rejects null hypothesis for 643 out of 768 components of differences vectors at the confidence level of 0.05. Thus, a vast majority of the CodeBERT features are sensitive to the bugfixing process.

VII. Details of the dataset collection procedure

Being most popular source code hosting resource, the GitHub is chosen as a source of data for the PyTraceBugs dataset. An initial list of 150 million GitHub public repositories is first fetched. Among this huge amount, approximately 150 thousands repositories are selected, whose prevailing language is Python.

Two basic ideas are employed to mine relevant source code. Namely, to get correct source code snippets, a refined version of the following idea is used. It is assumed that stable source code from well respected open-source GitHub repositories is more likely to be without bugs. As for the buggy source code, it is assumed that source code of the GitHub bugfix pull requests and commits from the top rated GitHub repositories is more likely to contain bugs, if those commits and pull requests are linked to special GitHub issues, having specific labels in repositories issues pages, *e.g*, "bug" or "error".

There are four levels of decisions made in the dataset collection process:

- choice of the GitHub repositories to be used as sources of the Python source code;
- selection of repositories issues to be analyzed for getting the bugfix code;
- selection of commits and pull requests to be utilized as sources of the bugfix code;
- choice criteria for the source code snippets to be used to guarantee source code correctness or defectiveness.

Below these selection criteria are discussed, first, for the part of the dataset, related to the buggy source code.

A. Collecting source code with bugs

1) Choice of repositories: To choose relevant repositories, the following principles are employed:

- issues page is available for a chosen repository;
- total count of issues is above 18, total count of pull requests is above 17, and the stargazer count is above 50 (these thresholds are 75th percentiles of counts values)
- repositories tags or releases count is above 2 (25th percentile)
- recent pull requests count over the half a year is above 26 (80th percentile).

Repository development in the GitHub is maintained by special web pages where the developers post reports on different issues/problems/bugs in the repository source code, which are called the issues pages. In well respected repositories, the issues pages are sources of standartized information about bugs in the repositories software, which allow their automatic processing.

The second selection criterion reflects overall repository popularity and its development activity. Moreover, tags/releases count threshold filters out those GitHub repositories, whose purpose is not for software development (*e.g.*, for web hosting, learning, or data storage). The last criterion selects repositories with high recent development activity.

Two first criteria selected about 10,000 repositories, whereas the last one filters about 2,200 repositories among them. Finally, repositories are filtered by stargazer count above 246 (50th percentile), which finally gives 1,100 repositories.

2) Choice of issues: Besides the issues page, each repository also maintains a labels page, in which a sort of categorization of possible types of repository issues is described. The standard practice in the GitHub for marking issues related to reporting bugs in the repository source code consists in associating these issues with special labels. This standartization allows mining snippets with bugs (see, *e.g.*, [7], [21]) by choosing only those issues that have a specific label. Unfortunately, the bug labels names can vary significantly among repositories. Therefore, the final list of criteria to select relevant labels has to be defined by a set of empiric rules of thumb based on the manual analysis of possible bug labels.

There is also a problem with selection of relevant issues using only bug label. Namely, applying this method, it is difficult to single out those issues, which report bugs at the level of snippets (*i.e.*, function or method implementations). Many repository bugs are at the higher level of library/project functionality. Since our research focus is on the low-level bugs, some other heuristics must be employed to refine selection of issues.

For the PyTraceBugs dataset, only those issues are selected, whose bug report contains a full error traceback report. The latter report lists a stack of calls of functions/methods. It ends up with a function or method, in which an error or exception occurs manifested by the corresponding error message. In our opinion, such issues definitely report low-level bugs. It is because:

- developers consider such errors/exceptions as bugs by labeling the issue suitably;
- the actual error occurs at the level of function or method.

Moreover, a selection is performed to remove issues, describing certain types of bugs. Namely, the following bugs are discarded: bugs located in dependent libraries (dependency bugs), bugs related to incompatibility with previous versions of the library (incompatibility bugs), and bugs related to porting parts of newer library version to older version (backport bugs).

This is done based on the issues labels. After applying all selection criteria, 8,165 issues are extracted.

For the selected repositories and their selected issues, the following information is collected:

- bug reports (issue titles and labels);
- full traceback reports and reproducing source code;
- the commits, pull requests, and merge commits (including their meta information, *e.g.*, titles, messages, states, *etc.*), referenced from issues timelines.

3) Choice of commits and pull requests: Source code changes related to handling repository issues are stored in the form of GitHub commits and pull requests. In addition to those changes, the commits and pull requests contain meta information annotating the made changes, *e.g.*, title, creation or merge timestamp, description, commit SHA1, pull request URL, *etc*. Among pull requests only merged ones are considered, which contain the code changes finally accepted by developers to be included into the repositories source code.

In the GitHub, there are several mechanisms of linking commits and pull requests to an issue, which they are aimed to handle. The information about linked commits and pull requests is available on issues web pages in their sections called the issues timelines. Four types of commits and pull requests exist:

- commits/pull requests, which close issues;
- commits referencing issues;
- pull requests mentioning issues;
- linked pull requests.

Closing commits and pull requests are highly relevant for the issue they close, *i.e.*, they contain fixes in the repository source code closely related to handling that issue. Referencing commits are usually also relevant.

There is also a mechanism in the GitHub, which allows one to close an issue when a pull request/commit is merged/committed. This mechanism prescribes for such commits and pull requests to contain a prescribed keyword and a reference in their message. This message should refer to the issue that is supposed to be closed, *e.g.*, “Fixes #123”. Such commits and pull requests are also relevant.

From the other hand, mentioning and linked pull requests may contain fixes for other issues, which are not directly related to the issue they mention. To filter out among such non-closing commits and pull requests, the meta information is used in selecting relevant ones. Below, our selection criteria are:

- commit/pull request should not reference more than 2 issues in its message;
- commit/pull request should contain a reference to issue in its message or title and if it occurs in the message, the prescribed tokens should precede this reference, *e.g.*, “closes”, “fixes”, *etc.*

When commits/pull requests treat many issues, it is difficult to select only those source code changes, which are relevant for a particular issue. The first selection rule discards such commits and pull requests.

In the second selection criterion, the relevance is enforced by providing explicit reference for issues in commits/pull requests titles or messages, similarly to the standard GitHub mechanism of closing commits/pull requests mentioned above.

Finally, those commits/pull requests are discarded, which contain irrelevant tokens, *e.g.*, “dependency” and “compatibility” in their titles and messages. This kind of selection is aimed to remove the code changes related to dependency, compatibility, and backport issues mentioned above. Applying all the aforementioned selection criteria, 6,013 pull requests and 2,908 commits are selected.

4) Choice of the code snippets: Finally, our selection criteria for the found snippets from bugfix source code are as follows:

- function/method name does not contain the “test” token;
- full filename for a snippet does not contain the “test” token;
- snippet is syntactically correct.

B. Collecting correct source code

1) Choice of repositories: To select top repositories as sources of the correct source code the following criteria are used:

- stargazer count is above 25;
- forks count is above 25;
- pull requests count is above 25.

As a result of applying those criteria, approximately 11 thousands of repositories are selected. These criteria are different from the filtering criteria used to get repositories being sources of bugfix source code. The main difference between them consists roughly in the following. First, in distinction to the criteria described above, availability of issues pages is also a selection criteria for the bugfix code. Second, filtering is tighter with respect to the stargazers and recent pull requests counts thresholds. This allows one to get bugfix source code of more or less high quality from the actively developing repositories.

2) Choice of the correct source code snippets: For correct source code a special type of stable source code is used from the selected repositories. In general, for top rated repositories, concepts of source code stability and correctness are close. Namely, developers being mostly highly qualified for such repositories tend to make changes in source code due to many reasons: fixing bugs, adding new features, improve readability and quality, and treating incompatibility and dependency issues. When the source code becomes of high quality, it becomes stable for a long time period that is specific for the repository it resides in. Here, there are some exceptions from the rule. For example, abandoned or unused source code tends to be stable.

In this work, the following heuristic is used to filter out relevant stable source code. A count of commits is computed for each snippet of source code, which is relative to the directory where this snippet is located. More specifically, it counts the number of the commits, which contain changes for at least one *.py* file (its file name does not contain the “test” token) in this specific directory and make no changes in that snippet. Our selection criterion consists in extracting snippets, whose corresponding count is above 100. This bound allows one to select about 11 million source code snippets (their total size is approximately 650 megabytes). In addition to this, selected snippets are filtered according to the same criteria as those from the subsubsection VII-A4. It gives about 5.7 million snippets.

To form the test sample, 170 snippets are chosen according to the following criterion from the obtained stable source code. A graph of function/method calls is computed within each repository. For each snippet (denote it by A) two groups of snippets are formed:

- the snippets, which call the snippet A in their implementation (these snippets correspond to incoming edges for A in the calls graph);
- the snippets, which are called from the implementation of A (these correspond to outcoming edges for A).

A random sample of 170 snippets is selected from those snippets, for which there are at least 3 incoming edges from other snippets, which themselves have at least 3 incoming edges. This allows us to protect additionally against the stalled code.

C. Computational cost of collecting the dataset and parallelization

Collecting the dataset is a time-consuming process because it requires downloading and processing large number of projects from the GitHub platform. Besides, the GitHub imposes a limit on frequency of the API requests, which further increases the required time. Collecting the dataset on a single machine would require about 8 days. To speed up the process, the Uran cluster [3] is used, which consists of nodes with two 18-core Intel Xeon Gold 6254 processors. Total list of repositories is split between the computing threads, with each of them processing its part of workload. Then, individual results from the threads are combined into a single one. By using 36 threads, the required time reduces to 6 hours.

VIII. Conclusion

In this work, a large labeled dataset is proposed intended for both training and evaluating of deep learning models for software bug prediction. It contains a number of examples of real bugs in the Python source code at the granularity of snippets, *i.e.*, implementations of functions or methods. It is split into training, validation and test samples, containing examples of buggy and correct code snippets. Confidence in labeling of the snippets in both training and validation samples is about 85% according to our estimates, whereas it is almost 100% for the test sample.

To further demonstrate quality of the dataset, a predictive model is built based on it, which classifies snippets into either correct or buggy. It predicts bugs with precision of 0.96 and recall of 0.34 on the manually validated test sample.

The dataset is available at https://github.com/acheshkov/pytracebugs.

[3]https://parallel.uran.ru

References

[1] M. Allamanis, H. Jackson-Flux, and M. Brockschmidt, "Self-supervised bug detection and repair," 2021. [Online]. Available: https://arxiv.org/abs/2105.12787

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017. [Online]. Available: https://arxiv.org/abs/1706.03762

[3] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," 2021. [Online]. Available: https://arxiv.org/abs/1812.08434

[4] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The ManyBugs and IntroClass benchmarks for automated repair of C programs," *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236–1256, 2015. [Online]. Available: https://doi.org/10.1109/TSE.2015.2454513

[5] 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*, 2014, pp. 437–440.

[6] 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*, ser. MSR '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 10–13. [Online]. Available: https://doi.org/10.1145/3196398.3196473

[7] R. Widyasari, S. Q. Sim, C. Lok, H. Qi, J. Phan, Q. Tay, C. Tan, F. Wee, J. E. Tan, Y. Yieh, B. Goh, F. Thung, H. J. Kang, T. Hoang, D. Lo, and E. L. Ouh, "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*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1556–1560. [Online]. Available: https://doi.org/10.1145/3368089.3417943

[8] D. A. Tomassi, N. Dmeiri, Y. Wang, A. Bhowmick, Y. Liu, P. T. Devanbu, B. Vasilescu, and C. Rubio-González, "BugSwarm: mining and continuously growing a dataset of reproducible failures and fixes," in *ICSE*. IEEE / ACM, 2019, pp. 339–349.

[9] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proceedings of the 2005 International Workshop on Mining Software Repositories*, ser. MSR '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 1–5. [Online]. Available: https://doi.org/10.1145/1083142.1083147

[10] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429–445, 2005.

[11] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*, 2007, pp. 9–9.

[12] C. Mills, J. Pantiuchina, E. Parra, G. Bavota, and S. Haiduc, "Are bug reports enough for text retrieval-based bug localization?" in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 381–392.

[13] S. Herbold, A. Trautsch, F. Trautsch, and B. Ledel, "Issues with szz: An empirical assessment of the state of practice of defect prediction data collection," 2020.

[14] M. Monperrus, "Automatic software repair: A bibliography," *ACM Comput. Surv.*, vol. 51, no. 1, Jan. 2018. [Online]. Available: https://doi.org/10.1145/3105906

[15] Z. Chen and M. Monperrus, "The CodRep machine learning on source code competition," 2018. [Online]. Available: http://arxiv.org/abs/1807.03200

[16] Z. Chen, S. J. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019. [Online]. Available: https://doi.org/10.1109/TSE.2019.2940179

[17] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," 2019. [Online]. Available: https://arxiv.org/abs/1812.08693

[18] R.-M. Karampatsis and C. Sutton, “How often do single-statement bugs occur? the ManySStuBs4J dataset,” in *Proceedings of the 17th International Conference on Mining Software Repositories*, ser. MSR ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 573–577. [Online]. Available: https://doi.org/10.1145/3379597.3387491

[19] E. N. Akimova, A. Y. Bersenev, A. A. Deikov, K. S. Kobylkin, A. V. Konygin, I. P. Mezentsev, and V. E. Misilov, “A survey on software defect prediction using deep learning,” *Mathematics*, vol. 9, no. 11, 2021. [Online]. Available: https://www.mdpi.com/2227-7390/9/11/1180

[20] J. Xu, F. Wang, and J. Ai, “Defect prediction with semantics and context features of codes based on graph representation learning,” *IEEE Transactions on Reliability*, pp. 1–13, 2020. [Online]. Available: http://doi.org/10.1109/TR.2020.3040191

[21] R. Ferenc, P. Gyimesi, G. Gyimesi, Z. Tóth, and T. Gyimóthy, “An automatically created novel bug dataset and its validation in bug prediction,” *Journal of Systems and Software*, vol. 169, p. 110691, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121220301436

[22] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, “CodeBERT: A pre-trained model for programming and natural languages,” 2020. [Online]. Available: https://arxiv.org/abs/2002.08155