

Python source code Analysis for Bug Detection using Transformers

Puneeth Batchu¹, Tanish Rohil Gali², Srujana Inturi³

^{1,2} Department of Computer Science and Engineering Chaitanya Bharathi Institute of Technology, Gandipet, Hyderabad, India

³ Assistant Professor, Department of Computer Science and Engineering, Chaitanya Bharathi Institute of Technology, Gandipet, Hyderabad, India

ABSTRACT: Effective bug detection is pivotal in software development, with the identification and localization of defects being crucial for robust applications. In Python-based programs, the conventional bug detection process relies on a Python interpreter, causing workflow interruptions due to sequential error detection. As Python's adoption surges, the demand for efficient bug detection tools intensifies. This paper addresses the challenges associated with bug detection in Python, focusing on the prevalence of built-in type bugs that can lead to code crashes.

Building on recent advancements, this survey explores bug detection methodologies across programming languages, emphasizing Python, JavaScript, and C. The diverse array of techniques covered includes static and dynamic analyses, machine learning-based bug detection, and predictive analysis engines (specifically deep-learning based). The survey provides insights into bug detection in Python programs, offering perspectives on addressing built-in type bugs and optimizing tools within the language's constraints.

KEYWORDS: Python, Built-in type bugs, Bug Detection, Static Analysis, Dynamic Analysis, Predictive Analysis

I. INTRODUCTION

In the realm of programming languages, Python has firmly established itself as a foundational tool, prized for its adaptability across diverse domains such as web development, data science, and machine learning. Despite its widespread use, Python, like any language, is not immune to bugs. These issues range from the common syntax errors to more intricate challenges associated with built-in types.

Built-in types in Python contribute significantly to the landscape of bugs developers encounter. These bugs can manifest as unexpected behaviors due to the language's flexibility with variables and data structures. Understanding and addressing built-in type bugs is crucial for maintaining code integrity and ensuring the reliable functionality of Python programs. The possible errors in python code are listed below in Figure 1.

The need for an advanced bug detection system in Python arises from the language's unique characteristics. Traditional bug detection methods, such as static and dynamic analyses, face challenges in effectively handling the interpretative nature of Python. Consequently, there is a growing demand for a systematic bug detection solution that can comprehensively analyze Python code, identifying both syntactic errors and subtle logical issues.

Transformers emerge as a promising solution for bug detection in Python. Unlike Long Short-Term Memory (LSTM) models, which have been employed for bug detection in the past, transformers demonstrate a notable advantage. LSTMs, while

effective in certain applications, exhibit limitations in capturing long-range dependencies within sequences. Optimized for sequential data processing, transformers employ self-attention mechanisms, excelling in capturing code semantics and understanding relationships within the code.

S.No.	Error type	Description
1.	IOError	is raised when I/O operator fails. Eg. File not found, disk full
2.	EOFError	is raised when, one of the file method i.e. read(), readline() or readlines(), try to read beyond the file.
3.	ZeroDivisionError	is raised when, in division operation, denominator is zero
4.	ImportError	is raised when import statement fails to find the module definition or file name
5.	IndexError	is raised when in a sequence - index/ subscript is out of range.
6.	NameError	is raised when a local or global name is not found
7.	IndentationError	is raised for incorrect indentation
8.	TypeError	is raised when we try to perform an operation on incorrect type of value.
9.	ValueError	is raised when a built in function/method receives an argument of correct type but with inappropriate value.

Fig. 1. Common errors in Python codebases

II. RELATED WORK

The study presents a feedback approach designed to aid novice programmers in the debugging process. The authors' objective is to pinpoint errors in a given code by analyzing its static structure. Beginning with an introductory section discussing debugging challenges faced by novice programmers and potential solutions, the study explores the implementation of tools such as syntax error highlighting in advanced code editors. The authors propose a bug identification methodology leveraging solutions and a language model based on LSTM networks. The model's architecture is

detailed, encompassing both LSTM networks and the proposed feedback mechanism. The study outlines methods for instruction, evaluation, and bug detection. Through the analysis of erroneous code samples gathered from the AOJ (Adaptive Online Judge), the authors elucidate the bug identification process within the model. Results are presented and scrutinized, including the identification of the most effective bug detection model and factors contributing to its limitations in certain scenarios. The report concludes by discussing potential avenues for future research, particularly the exploration of Bidirectional LSTM networks for bug identification. [1]

This research study suggests a technique to anticipate errors in software code using entropy-based metrics and neural network-based regression. The study compares the performance of this technique with statistical linear regression and analyses the relevance of bug prediction in software engineering. The study is broken into segments addressing the code update process, data collection and extraction procedure, results of statistical linear regression (SLR), results of neural network-based regression (NBR), threats to validity, and future work. The authors give references to important work and technologies employed in their research, bringing vital insights into bug prediction methodologies and their utility in software development [2].

The paper introduces Bugram, a novel bug detection technique that identifies potential weaknesses, inappropriate code usage, or unusual software behaviors using n-gram language models. Unlike traditional rule-based methods, Bugram employs a probabilistic approach to evaluate the likelihood of token sequences within a program, flagging sequences with low probabilities as potential issues. The authors evaluate Bugram on 16 open-source Java applications and compare its effectiveness to three other bug discovery tools based on graphs and rules. The results demonstrate Bugram's ability to detect flaws that might be overlooked by conventional methods, suggesting its potential as a supplementary approach to enhance software reliability. Furthermore, the research examines the utilization of n-gram models with different gram sizes, finding that 3-gram models are the most effective for bug discovery compared to other models. The authors also discuss potential future directions for Bugram, including its expansion to other programming languages and its integration with rule-based techniques to improve bug detection capabilities. [3].

This study introduces a predictive analysis engine in Python that captures execution records, encodes traces—including unexecuted branches—into symbolic constraints, and introduces symbolic variables to represent input values, their dynamic types, and feature sets for analyzing their variations. By addressing these constraints, the engine identifies flaws and the inputs that trigger them. The method proves particularly effective in examining complex real-world programs with numerous dynamic features,

owing to its clever encoding design which was based on the traces. The evaluation reveals that the approach uncovers 46 flaws from 11 real-world applications, including 16 previously unidentified vulnerabilities, demonstrating its utility as a tool for detecting bugs in Python systems. [4]

This document provides a comprehensive benchmark of 453 authenticated JavaScript faults derived from widely recognized JavaScript server-side apps. Each identified bug includes detailed information such as the bugs report, the test samples used to find it, and the corresponding patch for resolution. The benchmark offers complete access to both flawed and corrected versions of the programs, along with the ability to execute associated test cases. This facilitates highly reproducible empirical research and enables comparisons of JavaScript analysis and testing tools with ease. Additionally, the authors conducted both quantitative and qualitative evaluations to demonstrate the variety of defects present in the benchmark, underscoring its potential for software analysis and testing research in bugs prediction, and fault localization for JavaScript. This benchmark aims to fill the void of a centralized repository for JavaScript applications and faults, serving as a significant resource for the research community. [5].

This research introduces two datasets, ManyBugs and Intro-Class, comprising a total of 1,183 errors found in 15 C programs. These datasets are prepared to facilitate comparison of automated software repair algorithms and provide foundational experimental results for three established repair techniques. The authors argue that the absence of standardized benchmarks has hindered progress in automated software repair research, and they propose that their datasets can help address this issue. The document provides comprehensive details on the datasets' structure, including metadata files, test cases, and shell scripts for both white-box and black-box testing. Additionally, the authors discuss the classification of errors in the datasets, covering aspects such as issue priority, security relevance, and the types of modifications made in patches provided by developers. Additionally, the research provides initial experimental outcomes for current repair algorithms, such as GenProg, applied to both datasets. The authors anticipate that these datasets will facilitate qualitative analysis of program repair methodologies and encourage the development of new repair techniques. [6].

This paper introduces the EXCEPY Python benchmark, generated from 15 open-source projects present on GitHub. The benchmark, encompassing 180 faults in Python built-in types, seeks to reproduce and measure detection tools' performance. It solves limits in existing benchmarks, offering insights into real-world reproducible failures for tool evaluation and further study. The work evaluates problems to the benchmark's validity and offers future research pathways, leading to a comprehensive Python bug benchmark for researchers and practitioners. [7]

This paper represents an initial exploration into the application

of ML techniques for identifying defects in C programs. The authors examine the effectiveness of static program analysis in defect detection and the challenges associated with its use. They suggest leveraging ML methods to assist in categorization tasks, particularly when coupled with a large corpus of annotated codebase. Three off-the-shelf ML algorithms are employed along with a substantial corpus of programs available for training and evaluation. The authors compare the findings obtained through ML with those of the static program analysis tool which was internally used at Oracle. Although the initial findings show promise, further exploration indicates that the ML methods utilized are inadequate for static program analysis tools due to their low accuracy. [8].

In this research, BugAID is introduced as a novel approach for autonomously learning bug fixing change categories, focusing on language construct modifications in server-side JavaScript applications. Through an analysis of 105,133 changes across 134 JavaScript projects, the study identifies 219 common bug-fixing change types and examines 13 recurring issue patterns deemed to impose significant maintenance overheads, presenting potential candidates for automated tool assistance. Additionally, the study provides the BugAID toolkit and an empirical dataset, both freely accessible, facilitating the replication of the evaluation and empirical inquiry. [9].

This work focuses on the extraction and modelling of rich semantics and relationships from problem reports, that are critical for bug analysis tasks such as bug understanding, localization, and fixing. The authors suggest the use of knowledge graphs for representation of knowledge and management, since they adequately transmit complicated semantics in heterogeneous data. Entities and the relationships between them are crucial parts in the knowledge graph, with entities being the smallest knowledge unit and relations reflecting semantic interconnections between entities. The study analyses the challenges of extracting bug relations and entities from bug reports, such as the free-style and unstructured nature of the data, the diverse information types contained in bug reports, and the sparsity of relation features. The authors propose a hybrid model based on Bi-LSTM-RNN for extracting bug relations and entities. [10].

The study comprises bug fix analysis and problem cause categorization. The study presents the Orthogonal Defect Classification (ODC) approach, which studies defects in depth and assists with defect discovery, eradication, and prevention. The ODC approach has eight orthogonal defect attribute categories tied to codebase. The research also discusses the creation of a tree structure for representing bug fixes and a tree-based convolutional network for issue cause categorization. The objective is to classify bugs into separate bug cause groups depending on the link between the source of defects and bug repair. [11].

This article introduces a novel approach for spontaneously generating bug detectors based on names through ML. Unlike previous name-based bug detectors, DeepBugs distinguishes itself by analyzing identifier names using semantic representations, learning bug detectors instead of manual construction, exploring a broader range of bug patterns, and focusing on a dynamically typed programming language. The framework frames detection of bugs as a problem of classification and trains a classifier using examples of both correct and faulty code. This technology produces effective bug detectors capable of identifying various issues in real-world JavaScript code. DeepBugs offers a hopeful pathway to improve the efficiency and efficacy of problem identification in software development. [12].

This study introduces NP-CNN, a novel convolutional neural network designed for bug localization by learning common characteristics from both plain English descriptions and codebase in programming languages. NP-CNN utilizes lexical information as well as program structure data to enhance bug localization accuracy, surpassing existing techniques. The model consists of a lexical feature extraction layer and a cross-language feature fusion layer, enabling the extraction of semantic information from both lexical and program structural perspectives. Through experimental analysis conducted on well-known software projects, NP-CNN demonstrates significant advantages in automatically detecting troublesome codebase through bug reports.. [13]

This study introduces LS-CNN, a novel deep learning model aimed at enhancing bug identification by utilizing the sequential structure of codebase. This technique integrates LSTM and CNN architectures to spontaneously identify potential problematic codebase segments based on bug reports. By incorporating the sequential structure of codebase, LS-CNN surpasses limitations of previous bug localization methods, providing additional insights beyond lexical and structural information. The model's architecture involves utilizing CNN to extract semantic features from code statements while preserving their sequential integrity. Subsequently, LSTM is employed to capture the sequential relationships between statements in the context of program structure. LS-CNN demonstrates significant improvements over state-of-the-art techniques in localizing challenging files, offering a promising avenue for bug identification by effectively harnessing the sequential nature of codebase. [14]

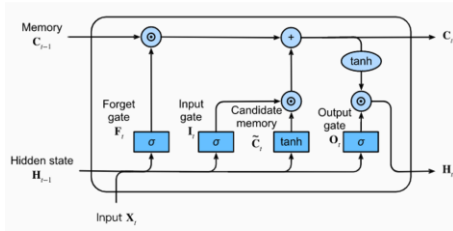
This work proposes HOPPITY, a learning-based tool for discovering and correcting defects in Javascript applications via graph transformations. By training on a massive dataset of bug-fixes from Github, HOPPITY leverages a graph neural network to identify patterns of proper and erroneous code, enabling it to effectively discover and patch faults in a range of systems. The authors demonstrate HOPPITY's effectiveness in discovering a wide range of faults, including functional concerns and refactoring modifications, and compare its performance with other state-of-

the-art tools, exhibiting superior accuracy and efficiency. Overall, HOPPITY offers a realistic and effective solution to preventing faults in production, particularly in the setting of sophisticated codebases. [15].

III. COMPARISON STUDY

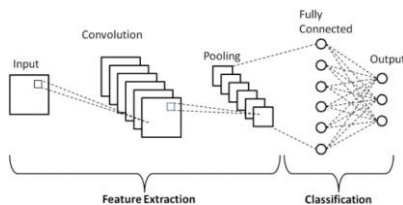
A. Long Short-Term Memory (LSTM)

LSTM networks stand as a significant breakthrough in deep learning, designed specifically to tackle the complexities of modeling sequential dependencies across various datasets. Unlike conventional recurrent neural networks (RNNs), LSTMs excel in capturing and preserving long-term dependencies across extensive sequences, making them highly adept for tasks such as NLP, time series analysis, and gesture recognition. The architecture of an LSTM incorporates specialized memory cells and gating mechanisms, enabling it to selectively store and retrieve information, thereby addressing challenges like the vanishing gradient problem commonly encountered in traditional RNNs.



B. Convolutional Neural Networks (CNN)

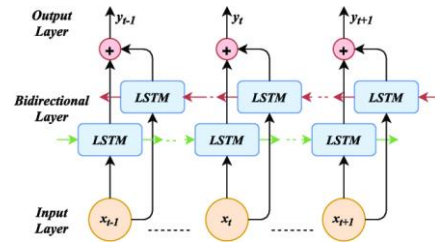
A CNN is a type of deep learning model which was designed to process data which is like grid, like images or videos. Convolutional layers apply filters to find patterns and features in the input data, pooling layers then work to reduce spatial dimensions, activation functions produce non-linearity, and fully connected layers learn complex correlations and generate predictions. CNNs stack these components and use techniques like backpropagation to determine the optimal weights and biases during training to minimize a given loss function. CNNs have revolutionized tasks like object detection and image recognition, making them an indispensable tool in the domain of Image processing & CV and beyond.



C. Bidirectional Long Short-Term Memory (BiLSTM)

BiLSTM networks mark a notable stride forward within the domain of deep learning, particularly in managing sequential data.

Unlike conventional LSTMs, BiLSTMs introduce bidirectional processing, empowering them to grasp contextual cues from both preceding and subsequent inputs. This bidirectional functionality proves advantageous in tasks demanding a deep understanding of temporal dependencies, including speech recognition, sentiment analysis, and named entity recognition. The BiLSTM architecture incorporates two interconnected LSTM layers—one processes the input sequence forwards, while the other does so backwards—facilitating the network in efficiently assimilating information from both temporal directions.



D. Decision Trees

Decision Trees are a foundational machine learning algorithm, serving as interpretable and potent tools for classification and regression tasks. Their intrinsic structure involves a sequence of binary decisions, depicted as a tree-like arrangement, enabling intuitive visualization and understanding of decision-making procedures. The algorithm iteratively divides the input space using features, generating nodes signifying decision points and leaves indicating ultimate outcomes. Decision Trees excel in situations where the connections between input features and target variables are intricate or non-linear.

E. Random Forest (RF)

Random Forest stands as a robust ensemble learning method in machine learning, versatile for both regression and classification tasks. In its training phase, it generates numerous decision trees by choosing a random subset of the training data and a random selection of the characteristics. These individual trees' predictions are then combined to produce the final prediction. Generally, classification tasks utilize the mode, while regression tasks employ the mean. Random Forest mitigates overfitting through feature selection and data randomization, resulting in a dependable and precise model. Its standout features encompass superior prediction accuracy, adept management of high-dimensional data, resilience against overfitting, and the ability to evaluate feature importance, cementing its widespread recognition in the field.

IV. COMPARISON ANALYSIS

Recurrent models such as LSTM networks are recognized for their capacity to grasp sequential dependencies within data, rendering them apt for discerning the intrinsic organization of code. However, LSTMs may struggle with their computational intensity, especially with large datasets. Convolutional Neural Networks

(CNNs) are adept at capturing local patterns and hierarchical representations in code but might lack the capacity to capture long-term dependencies. Bidirectional LSTM Recurrent Neural Networks (Bi-LSTM-RNN) attempt to address this limitation by capturing dependencies in both forward and backward directions, enhancing their ability to understand code intricacies. On the other hand, Random Forest, an ensemble method, provides robust predictions by combining multiple decision trees. It offers faster training times and is less sensitive to the need for large datasets, making it a pragmatic choice for bug detection, particularly when computational resources are limited. The interpretability of Random Forest can also be advantageous, aiding in understanding the reasoning behind bug predictions.

The selection among these algorithms involves trade-offs. For instance, deep learning models like LSTM and Bi-LSTM-RNN excel in capturing sequential and long-range dependencies but may demand substantial computational resources. CNNs leverage local patterns but may not capture long-term dependencies as effectively. Random Forest, while computationally efficient and interpretable, may have limitations in modeling intricate code relationships. The selection should be in line with the precise needs of the bug detection task, taking into account factors like dataset magnitude, interpretability, and the availability of computational resources. Practical experimentation and fine-tuning are essential to determine the algorithm that best suits the characteristics of the codebase and the bugs targeted for detection.

V. CONCLUSION

The research offers a comprehensive overview of contemporary methodologies of bug detection, spanning various aspects of software development. It begins by emphasizing the importance of assisting novice programmers through feedback systems and leveraging LSTM-based approaches for problem identification, showcasing the potential of ML to streamline coding processes and boost productivity. Following this, diverse bug detection techniques are explored, ranging from entropy-based metrics to probabilistic n-gram LM's and symbolic constraint encoding for trace-based recognition in Python code. These studies underscore the diversity of methods employed across different programming languages.

Furthermore, the research highlights the significance of benchmarking endeavors and the implementation of innovative bug analysis tools, such as knowledge graphs, orthogonal defect classification, and DL models like NP-CNN and HOP-PITY. These tools leverage conceptual representations, exploit the sequential nature of codebase, and utilize graph transformations to improve accuracy and efficiency in bug detection. Together, these discoveries play a pivotal role in enhancing bug identification processes, revealing effective strategies that

harmonize traditional software engineering principles with cutting-edge advancements in machine learning, ultimately bolstering program reliability.

REFERENCES

1. Y. Teshima and Y. Watanobe, “Bug detection based on lstm networks and solution codes,” in *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pp. 3541–3546, 2018.
2. A. Kaur, K. Kaur, and D. Chopra, “Entropy based bug prediction using neural network based regression,” in *International Conference on Computing, Communication Automation*, pp. 168–174, 2015.
3. S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan, “Bugram: Bug detection with n-gram language models,” in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 708–719, 2016.
4. Z. Xu, P. Liu, X. Zhang, and B. Xu, “Python predictive analysis for bug detection,” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016*, (New York, NY, USA), p. 121–132, Association for Computing Machinery, 2016.
5. P. Gyimesi, B. Vancsics, A. Stocco, D. Mazinanian, Beszédes, R. Ferenc, and A. Mesbah, “Bugsjs: a benchmark of javascript bugs,” in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pp. 90–101, 2019.
6. C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu,
7. S. Forrest, and W. Weimer, “The many bugs and introclass benchmarks for automated repair of c programs,” *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236–1256, 2015.
8. X. Zhang, R. Yan, J. Yan, B. Cui, J. Yan, and J. Zhang, “Excepy: A python benchmark for bugs with python built-in types,” in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 856–866, 2022.
9. T. Chappelly, C. Cifuentes, P. Krishnan, and S. Gevay, “Machine learning for finding bugs: An initial report,” in *2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, pp. 21–26, 2017.
10. Q. Hanam, F. S. d. M. Brito, and A. Mesbah, “Discovering bug patterns in javascript,” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016*, (New York, NY, USA), p. 144–156,

Association for Computing Machinery, 2016.

11. D. Chen, B. Li, C. Zhou, and X. Zhu, “Automatically identifying bug entities and relations for bug analysis,” in *2019 IEEE 1st International Workshop on Intelligent Bug Fixing (IBF)*, pp. 39–43, 2019.
12. Z. Ni, B. Li, X. Sun, T. Chen, B. Tang, and X. Shi, “Analyzing bug fix for automatic bug cause classification,” *Journal of Systems and Software*, vol. 163, p. 110538, 2020.
13. M. Pradel and K. Sen, “Deepbugs: A learning approach to name-based bug detection,” *Proc. ACM Program. Lang.*, vol. 2, oct 2018.
14. X. Huo, M. Li, and Z.-H. Zhou, “Learning unified features from natural and programming languages for locating buggy source code,” in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI’16*, p. 1606–1612, AAAI Press, 2016.
15. X. Huo and M. Li, “Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code,” in *Proceedings of the 26th International Joint Conference on Artificial Intelligence, IJCAI’17*, p. 1909–1915, AAAI Press, 2017.
16. E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang, “Hoppity: Learning graph transformations to detect and fix bugs in programs,” in *International Conference on Learning Representations*, 2020.