



Exploring Name-based Bug Detection in Python

by

Subrata Das

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE FACULTY OF GRADUATE STUDIES
OF LAKEHEAD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

2024

Lakehead University
Thunder Bay, Ontario, Canada

Examining Committee Membership

The thesis of Subrata Das, Exploring Name-based Bug Detection in Python, is approved:

Supervisor:

Dr. Muhammad Asaduzzaman
Assistant Professor, School of Computer Science,
University of Windsor, Windsor, Ontario, Canada

Supervisor:

Dr. Salimur Choudhury
Associate Professor, Department of Computer Science,
Queen's University, Kingston, Ontario, Canada

Internal Examiner:

Dr. Garima Bajwa
Assistant Professor, Department of Computer Science,
Lakehead University, Thunder Bay, Ontario, Canada

External Examiner:

Dr. Zubair Md Fadlullah
Associate Professor, Computer Science,
Western University, London, Ontario, Canada

ABSTRACT

Names of source code elements provide useful contextual information about the code and development tasks. Prior studies leverage the similarity between the names of arguments and method parameters to detect bugs that are caused by accidentally swapping arguments while calling methods. This requires establishing the mapping between method calls and their definitions. However, it is a challenging task to establish the mapping because of the complexity involved with the process (e.g., missing external libraries). This thesis aims to understand the performance of name-based argument-related bug detection techniques in Python, a popular general-purpose, statically typed programming language.

Towards this direction, this thesis conducts a study that first investigates the similarity between arguments and their method parameters in Python code. The above step follows by establishing the mapping of method calls to their definitions and evaluating the performance of existing name-based techniques to detect swapping argument-related bugs in Python. Finally, a technique has been developed that uses argument usage patterns and expression types in source code with name-based similarity matching to improve the performance of detecting argument-related bugs. Evaluation of the proposed technique with a large collection of open-source Python projects shows that the technique can detect argument-related bugs with high accuracy even when the method definitions are missing. One potential solution to prevent argument-related bugs from occurring is to use code completion. An argument recommendation system suggests method arguments as a developer types the code. Thus, the second part of the thesis focuses on completing arguments of method calls. In particular, this thesis investigates the efficacy of large language models in recommending arguments for API (Application Programming Interface) method calls.

ACKNOWLEDGEMENTS

At the forefront of my gratitude, I extend sincere appreciation to my supervisors, Dr. Muhammad Asaduzzaman and Dr. Salimur Chowdhury, whose unwavering guidance, insightful recommendations, encouragement, and remarkable patience have been instrumental in the fruition of this thesis. Their indispensable support has been pivotal, without which this endeavor would not have come to fruition.

I would like to express my gratitude to Dr. Zubair Fadlullah (Western University) and Dr. Garima Bajwa (Lakehead University) for their contributions to my thesis committee. Their guidance, evaluation, and insightful remarks helped this thesis to be completed successfully.

I am also thankful and express my gratitude to the Department of Computer Science, Lakehead University for their kind assistance in the form of scholarships, awards, and bursaries, which allowed me to focus intently on my thesis work. I want to express my gratitude to all of my friends and fellow Department of Computer Science staff who have supported me along this journey. I am especially grateful to Kawser Wazed Nafi who provided constant motivation, brilliant ideas, and expert advice to shape my ideas. I also convey my gratitude to my lab mate and coworker, MD Anaytul Islam, as we worked together to understand the modern field of software engineering. I thank Dr. Sheikh Moniruzzaman, Pronab Ghosh, and Shouvik Paul for their prudent guidance.

I am deeply grateful to my friends in Thunder Bay for their unwavering support and encouragement throughout my endeavors. I extend special thanks to Shofi Ahmed, Sakib Ali, Waishari Das, Mostansir Nayeem, Prithu Aldrin Costa, Hasib Kamal, Nahiyah Al Mahmood, Ragib Raonok, and Khandaker Billalur Rahaman for their invaluable friendship and assistance. I am grateful to everyone for their participation in this thesis, even if they are not named specifically. Your participation has been really helpful to my growth, both personally and professionally.

This thesis is dedicated to my beloved father, Mr. Subodh Kumar Das, and mother, Mrs. Kakali Biswas, whose unwavering inspiration and steadfast support have been the guiding lights illuminating every step of my journey, both academic and personal.

Contents

Abstract	iii
Acknowledgements	iv
Table of Contents	v
List of Tables	ix
List of Figures	xi
1 Introduction	1
1.1 Motivation	1
1.2 Research objectives and significance	5
1.3 Contributions of the Thesis	6
1.4 Outline of the Thesis	7
2 Related Work	8
2.1 Name-Based Bug Detection for JavaScript	8
2.2 Name-Based Source Code Analysis and Bug Detectors	9
2.3 Parameter-Argument-Related Bug Detectors	12
2.4 Usage Pattern-Based Source Code Analysis	15
2.5 Deep Learning for Bug Detection	17
2.6 Usage of Large Language Models	18
2.7 Conclusion	19
3 Linking Method Calls to their Definitions	21
3.1 Introduction	21
3.2 Dynamic Type Solving of Variables in Python	21
3.3 Linking Method Calls to their Definitions	25
3.3.1 Method Definition Pattern	26
3.3.2 Method Call Patterns	27
3.3.3 Analysis 1: Method Call and Method Definitions in the Same File	28

3.3.4	Analysis 2: Both an import statement and method call sequence (Outside the same file Method Call Mapping)	31
3.4	Example of Resolving Method Call	32
3.5	Result of Type Detection:	34
3.6	Evaluation Procedure:	35
3.6.1	Experimental Setup	35
3.6.2	Evaluation Metrics	35
3.7	Result of Mapping Algorithm	35
3.8	Conclusion	36
4	Exploring Name-based Bug Detection in Python	38
4.1	Introduction	38
4.2	Background	41
4.3	Data Collection	44
4.4	Methodologies	45
4.4.1	Data Extraction and Generalization	45
4.4.2	Linking Method Calls to its Definitions	46
4.4.3	Binder Generation	46
4.4.3.1	Binder Generation for Correct Code Pattern	46
4.4.3.2	Swapping Argument Sequence for Wrong Code Pattern	53
4.4.4	Context Collection for Word2vec Model	53
4.4.4.1	Context Collection for DeepBugs Model(DBM)	54
4.4.4.2	Context Collection for API Usage-Context Mode(AUM)	55
4.4.4.3	Context Collection for Argument Usage Pattern with Parent Information Model(AUCM Model)	55
4.4.4.4	Context Collection for Argument Usage Pattern with Parent Information and Expression Type Information(AUCMET)	55
4.5	Experimental Setup	57
4.6	Evaluation Procedure	58
4.6.1	Evaluation Metrics	59
4.6.2	RQ1: How do programmers use argument and parameter in Python?	59
4.6.2.1	Distribution of Lexical Similarity between a Parameter and its Arguments in Python?	60
4.6.2.2	What is the Length of Arguments and Parameters in Python?	62
4.6.2.3	What are the reasons for the dissimilarity of method argu- ments and their corresponding parameters?	64
4.6.2.4	Can we filter out the arguments that have lower similarity values with their arguments?	65

4.6.3	RQ2: Effectiveness of the Proposed Technique	65
4.6.4	RQ3: Impact of Different Source of Information	67
4.6.5	RQ4:Performance Comparison Of AUCMET With Pre-Trained Code- Bert	71
4.6.6	RQ5: Efficiency of the Proposed Technique	72
4.6.7	Additional Analysis To Evaluate The Performance of AUCMET . .	72
4.6.7.1	Performance of DeepBugs on Python and AUCMET based on Expression Type:	73
4.6.7.2	Performance of DeepBugs on Python and AUCMET based on Context Length	74
4.6.7.3	Performance of DeepBugs on Python and AUCMET based on Method Call Appearance in Training Examples	76
4.7	Threats to Validity	77
4.8	Conclusion	77
5	An Empirical Study of Argument Recommendation by LLM in Python	79
5.1	Introduction	79
5.2	Background	80
5.2.1	Statistical Language Models	80
5.2.2	Code Completion in Python	81
5.2.3	Argument Recommendation	81
5.2.4	Usage of Large Language Model	81
5.3	Research Significance	82
5.4	Dataset	83
5.5	Approach	83
5.5.1	Data Extraction and Preprocessing	84
5.5.1.1	Method Call Extraction	84
5.5.1.2	Global Variable Extraction	84
5.5.1.3	Determine The Scope Of The Method Call	85
5.5.2	Context Collection	85
5.5.3	Model Description And Using For Argument Generation	87
5.5.3.1	Input Generation for Models for Evaluation:	91
5.6	Evaluation Procedure	92
5.6.1	Evaluation Metrics	92
5.6.2	Performance Comparison of CodeT5, CodeBERT, Code Llama Mod- els	93
5.6.3	Expression-wise Performance Comparison of CodeT5, CodeBERT, Code Llama Models	94

5.6.4	Argument Precedence based Performance Comparison of CodeT5, CodeBERT, Code Llama Models	96
5.7	Result	97
5.8	Taxonomy of Argument Types For Future Research	97
5.9	Conclusion	98
6	Conclusion	100
6.1	Summary	100
6.2	Future Work	101
A	Installation of Modules and Environment Setup	103
A.1	Installation and Update Ubuntu	103
A.2	Required Modules	103
A.3	Parsing With Python AST	105
A.4	Downloading Projects from GitHub	109
A.5	Reproducing the study Exploring Name-based Bug Detection in Python . .	109
A.6	Reproducing the study Empirical Study of Argument Recommendation by LLM in Python	110
	Bibliography	111

List of Tables

Table 3.1	Description of Updating the Knowledge Base for Type Detection . . .	24
Table 3.2	Type Collection from Figure 3.2	25
Table 3.3	Classification of Expression Type Extraction	26
Table 3.4	Expression Type of Value of Assignment	34
Table 3.5	Manual Investigation Report of Mapping of Method Calls and their Definitions (250 examples for each project)	36
Table 4.1	Generalization of Extracted Argument Names	43
Table 4.2	Binder Information for DeepBugs Model(DBM) and API UsageBased Model (AUM)	47
Table 4.3	Binder Information for AUCM Model	48
Table 4.4	Collected Context for AUCM Model	49
Table 4.5	Collected Context for AUCMET	51
Table 4.6	Collected Context for Latest and Parent Block Usage Context with Expression type information (Contd.)	52
Table 4.7	Encoding of Tokens for AUCMET	54
Table 4.8	Performance Comparison with DeepBugs Approach and AUCMET For Mapped Method Calls	66
Table 4.9	Performance Comparison with DeepBugs Approach and AUCMET For All Method Calls	67
Table 4.10	Combination Of Different Source Information Of AUCMET	68
Table 4.11	Analysis of Different Sources of Information	69
Table 4.12	Performance Comparison of DeepBugs, AUCMET, Pre-trained Large Language Model-BERT	71
Table 5.1	Collected Context for Expression-based Analysis and Overall Perfor- mance Analysis	86
Table 5.2	Expression Types With Their Examples In Python	88
Table 5.3	Expression Types with their examples in Python(contd.)	89
Table 5.4	Over All Model Performance Model Performance	94
Table 5.5	Expression wise- Model Performance	95

Table 5.6 Model Performance-Precedence-wise Result 97

List of Figures

Figure 1.1	Equally Typed Parameters	1
Figure 1.2	Swapping Argument Related Bugs	2
Figure 1.3	Library Mapping of Method Call	4
Figure 3.1	Example of Dynamic Type	22
Figure 3.2	Example of Dynamic Type(contd.)	23
Figure 3.3	Different Types of Method Calls	27
Figure 3.4	Method Call Pattern	28
Figure 3.5	Method call and its Corresponding Definition in the Same file	29
Figure 3.6	Example of self as a Receiver of A Method Call	30
Figure 3.7	Import Statement Analyzer	31
Figure 3.8	Usage of Import Statement Analyzer to Generate Binder for DeepBugs	31
Figure 3.9	Example of Mapping a method call to method definition from another file	33
Figure 4.1	Method Call and its Mapped Definition	39
Figure 4.2	Number of Examples Covered by Argument Count 1,2,3	44
Figure 4.3	Encoding of A Word Based on Different Expression Type	56
Figure 4.4	Comparison of Encoding For Other Programming Language	57
Figure 4.5	Work Flow Diagram	58
Figure 4.6	Lexical Similarity between Argument and its corresponding Parameter	59
Figure 4.7	Length of Argument Name by Characters	60
Figure 4.8	Length of Parameter Name by Characters	61
Figure 4.9	Length of Parameter Name by Terms	62
Figure 4.10	Length of Argument Name by Terms	63
Figure 4.11	Argument Average Similarity by Characters and Terms	64
Figure 4.12	Parameter Average Similarity by Characters and Terms	64
Figure 4.13	Performance of Both DeepBugs Model and AUCMET	67
Figure 4.14	Accuracy of Expression Type-based Analysis (20<=Frequency)	73
Figure 4.15	Accuracy of Expression Type-based Analysis (20<=Frequency)	74

Figure 4.16 Effect of Context Length on Performance For DeepBugs and AUCMET	75
Figure 4.17 Average Method Call Frequency in Training Set Case	76
Figure 5.1 An Example of Code Completion in Visual Studio Code and PyCharm IDE	80
Figure 5.2 An Example of Source Code for Context Collection	82
Figure 5.3 Input Context Generation for Argument Generation Based Study with CodeBert, Code Llama, CodeT5	87
Figure 5.4 Input Context Generation for Argument Precedence Based Study with CodeBert, Code Llama, CodeT5	90
Figure 5.5 Categorization of Expression Type	98
Figure A.1 Parsing With Python AST	106
Figure A.2 Showing Instances of a Node	107
Figure A.3 Visiting A Parsed Node	108
Figure A.4 Visiting A Parsed Node by Class method	108
Figure A.5 Result of Parser	109
Figure A.6 Process of Git Repository Clone	110

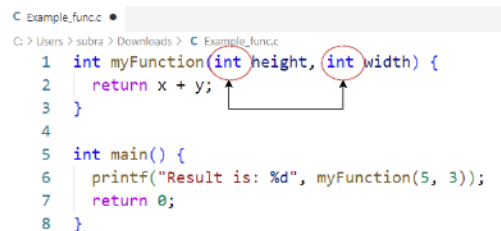
Chapter 1

Introduction

This chapter provides a brief introduction to the thesis and explains the significance of the work. Section 1.1 describes the motivation for this thesis. The problem statement of this thesis is delineated in Section 1.2. Section 1.3 explains the contributions of the thesis. Finally, Section 1.4 provides an outline of the remaining chapters.

1.1 Motivation

Name-based source code analysis lies in how effectively the semantic meaning of identifier names, such as variables, functions, classes, etc., can be used for different software engineering tasks, such as the semantic representation of source code, bug detection, and type prediction. Identifier names provide useful information about the pattern of a source code. These identifier names are declared by the developers, which projects the understanding and increases the readability of source code [1]. As an example, consider a code fragment containing the following five different tokens: “sum”, “x”, “y”, “return” and “count”. Therefore, the appearance of these names in a code fragment as tokens will define the fact that the source code may return a value that can summate two variables. The names of iden-



```
C Example_func.c •
C: > Users > subra > Downloads > C Example_func.c
1 int myFunction(int height, int width) {
2     return x + y;
3 }
4
5 int main() {
6     printf("Result is: %d", myFunction(5, 3));
7     return 0;
8 }
```

Figure 1.1: Equally Typed Parameters

tifiers are static properties. Therefore, popular static tools are widely used based on source code context. This name-based approach can detect bugs in dynamically typed languages

like JavaScript [2]. For the equally typed variables, this name-based analysis provides extra information about the variables. As an example, in Figure 1.1, the method call “myFunction” has two parameters, “height”, “width” and both of them have the same type. As a result, it is hard to detect any argument-related bugs from their type information. However, the name-based analysis can provide extra information about the parameters. Therefore, the name-based approach can be used for dynamically typed languages like Python and R.

```

1 import pandas as pd
2 data = {
3     'Student ID': [101, 102, 103, 104, 105, 106, 107, 108, 109, 110],
4     'Marks': [88, 92, 85, 74, 91, 78, 84, 89, 95, 87],
5     'Student Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eva', 'Fiona', 'George', 'Hannah', 'Ian', 'Jack']
6 }
7 df = pd.DataFrame(data)
8 def student_mark_check(df, student_id, student_name):
9     df_temp=df[df['Student ID'].apply(int)==student_id]
10    df_temp=df_temp[df_temp['Student Name']==student_name]
11    return df_temp['Marks'].to_list()
12 print(student_mark_check(df, 101, "Alice")) Correct Argument Sequence
13 print(student_mark_check(101,df, "Alice")) Wrong Argument Sequence

```

```

Run: Example_check x
C:\Users\subra\anaconda3\envs\New_home_Starting\python.exe C:\Users\subra\PycharmProjects\New_home_Starting\Example_che
[88]
Traceback (most recent call last):
  File "C:\Users\subra\PycharmProjects\New_home_Starting\Example_check.py", line 15, in <module>
    print(student_mark_check(101,df, "Alice"))
  File "C:\Users\subra\PycharmProjects\New_home_Starting\Example_check.py", line 11, in student_mark_check
    df_temp=df[df['Student ID'].apply(int)==student_id]

```

Figure 1.2: Swapping Argument Related Bugs

This thesis focuses on Python, the most popular programming language, which is used massively to build new software systems [3]. Python is a dynamic programming language with imperative, logical, functional, and object-oriented features [4]. Name-base string similarity and neural network-based embeddings of identifiers are mostly used to determine the similarity between the names of identifiers, which generates a group of words of the same meaning—converting the word to a similar vector space to preserve the meaning and mapping to the identifiers which have an almost similar meaning. Besides, name-based analysis is commonly used to fix faults by gathering information from the naming convention of the identifiers. It provides a subtle understanding of how the name of an identifier provides an important aspect of code structure that helps a programmer debug the code. Other than that, a programmer can use the name-based analysis to generate lexical similarity [5], which will add context-based information based on the usage of an identifier. With the name of an identifier, programmers can add different properties, such as type information, the behavior of the identifier, the location of the declaration, usage of an identifier, the original meaning of the identifier, and how it is used in the source code can be used in name based analysis.

Passing the correct argument to a method call is mandatory to execute a program and generate the expected output. Calling a method multiple times is common in source code development. The bugs related to a method call are mainly two types- **incorrect naming of the method call and incorrect passing of arguments to the method call**. The name of the method call and declaration must follow the same name while calling the method. As the method call is always related to a method definition, there is a slight chance of missing or misspelling the name of the method call at the time of call. Our analysis of method call mapping showed that programmers are less likely to misspell the name of a method. The second concern for a correct API call is the bug related to the method arguments. The argument-related bugs are- passing wrong arguments, passing arguments in the wrong sequence, and passing more arguments than required while calling a method. A prior study on the argument-related bug focused on incorrectly swapping two adjacent arguments drew our interest in studying the swapping argument-related bugs in Python. For dynamically typed programming languages, it is difficult to determine the exact variable type, and while passing the variable as an argument, programmers may select the variable correctly but in the wrong sequence. In Figure 1.2, at line 12 we found a method call, “student_mark_check” which will expect the first argument, “df” which is a pandas Dataframe and the second argument is “101” which is an integer. The incorrect sequence is (101, df, “Alice”). The IDE (Integrated Development Environment) did not show any error or warning while writing the code. However, the IDE showed the following error: “TypeError: “int” object is not subscriptable” while executing the code. Therefore, while calling a method, the compilers are not getting the type information for the variables, and if they are accidentally swapped, they cannot warn the programmers. The extracted name contexts are treated as patterns. The existing name-based analysis only considered the identifier names and their local context (see section -4.2). Therefore, this context of a particular token suffered from the inconsistency of name and value as the local context does not contain all the information for a token. This inconsistent data leads to false prediction of a deep learning model. The swapping argument-related bug detection techniques used the type of arguments to detect the swapping argument-related bugs. Though the model worked effectively for statically typed languages, such as C, C++, and Java, there needed to be evidence of performance for dynamically typed languages, such as Python and JavaScript. The study of Pradel et al. [2] used the static type information from source code to build a bug detector for JavaScript known as DeepBugs, which used 30% type information of the variables from existing JavaScript source code. Meanwhile, in Python, the ratio of static type is less than 10% for a large project (which contains 10k method calls). Therefore, the existing swapping bug detection approach will not be effective for detecting argument-swapping-related bugs for projects with more library method calls and definitions. Again, the definitions of method calls can be located in different files or external libraries. Existing

approaches have yet to try to bring the method definition information from third-party libraries. In Figure 1.3, the “join” method does not have a method definition in the same file. Therefore, to solve the method call mapping issues, we proposed an algorithm that successfully mapped 80% of the method calls from both library and project method calls. For type detection of variables, there were 40% call expressions of value. This mapping will bring the method body, and the variable type will be detected from the return statement. Therefore, body analysis and method mapping solve the dynamic type allocation.

```

download.py × ntpath.py
download.py > _download
346 def _download(name):
367     if total_parts > 1:
368         concatenated_folder_name = "{fname}.gz".format(fname=name)
369         concatenated_folder_dir = os.path.join(tmp_dir, concatenated_folder_name)
370         for part in range(0, total_parts):
371             url_data = "{base}/{fname}/{fname}.gz_0{part}".format(base=DOWNLOAD_BASE_URL
372
373             fname = "{f}.gz_0{p}".format(f=name, p=part)
374             dst_path = os.path.join(tmp_dir, fname)
375             urllib.urlretrieve(
376                 url_data, dst_path,
377                 reporthook=partial(_progress, part=part, total_parts=total_parts)
378             )

download.py × ntpath.py
C: > Users > subra > AppData > Local > Programs > Python > Python310 > Lib > ntpath.py > isabs
100
101
102 # Join two (or more) paths.
103 def join(path, *paths):
104     path = os.fspath(path)
105     if isinstance(path, bytes):
106         sep = b'\\'
107         seps = b'\\/'
108         colon = b':'
109     else:
110         sep = '\\'
111         seps = '\\/'
112         colon = ':'

```

Figure 1.3: Library Mapping of Method Call

We found a few studies on swapping argument-related bugs in Python from the above-mentioned issues. Type detection for a variable faces complexity due to the maximum detected value of variables being method calls, and as Python is a library-based programming language, definitions of method calls can be found anywhere in the environment. A mapping algorithm can resolve those method calls and bring out exact types from method definitions. This provides additional context information for variables and supports the model for detecting swapping argument-related bugs in Python. When a study showed that 70% of the method calls were from external libraries or built-in methods, we worked with usage context-based analysis. We found that the method argument is declared before it is used for the dynamic programming language, regardless of the change of a variable.

Therefore, the pattern-based study showed that if the source code matches other source codes, it must follow a pattern. Therefore, we got motivated to extract the name-based information and merge it with the local context to provide extended information. Our model performed 10% more accurately than the existing model [2] (see the model description for Python at -4.1). Again, this model faced large vocabulary issues and token usage ambiguity. Similar to the embedding type information with the name-based analysis, we embedded the expression type information, which performed better than all models.

While the first study of this thesis focuses on detecting bugs caused by incorrectly ordered method arguments, the second study focuses on understanding the efficacy of pre-trained models in recommending method arguments. Most pre-trained models are designed to understand the context of source code and generate source code. Though those models generated code segments from a given context, our study verifies whether these models can generate arguments correctly or not.

1.2 Research objectives and significance

The most relevant study to this thesis was done by Pradel et al. [2] [6] for JavaScript. They proposed a technique, called DeepBugs, that uses a machine learning model for detecting incorrectly ordered argument-related bugs leveraging names in source code. Instead of relying on manual coding, it uses a semantic representation to acquire bug detectors autonomously. The bug detection process is cast as a binary classification problem, where a classifier is trained to differentiate between correct and incorrect code. To address the difficulty of acquiring diverse examples of correct and incorrect code for optimal learning, incorrect code instances are generated by applying basic transformations to an existing code corpus. A significant discovery from this work is that bug detectors trained on artificially introduced bugs demonstrate effectiveness in detecting real-world bugs.

The preliminary objective of this thesis is to check the performance of the same process for another dynamic programming language, Python. While applying the same approach to the Python dataset, the lack of method definitions for the corresponding method calls and lack of information on argument types are faced due to the severely dynamic nature of Python. Therefore, our first approach is to resolve the mapping of method calls to their definitions and gather as much information as possible for the deep-learning model. However, the mapping algorithm brings out approximately 30% to 35% more examples than the original DeepBugs Model (on JavaScript), and the maximum number of method calls still needs to be explored. To mitigate this problem, this thesis proposed an approach to detect argument-swapping-related bugs for Python by analyzing the argument usage pattern, which is effective when the method definitions are missing.

Therefore, this thesis focuses on mapping method calls to their corresponding method

definitions, providing a warning if the sequence of the method arguments is incorrect. While studying the approach, we observe the importance of vector generation from a given context. Therefore, this thesis conducts an empirical study of how large language models perform in the case of generating correct arguments for Python. We used three popular pre-trained large language models to generate the arguments based on source code context. We studied manually by collecting random samplings from those method calls. The study shows evidence of the high efficiency of a large language model for some of the argument expression types. These pre-trained models are well-trained by a large code corpus to understand the contextual embedding of tokens with long-range dependencies. This provides support for generating models to detect bugs [7] or generating a new segment [8]. In most cases, the pre-trained model performs better than other deep neural networks, for example- Recurrent Neural Network (RNN) and BiLSTM-based methods. Our second study focused on the performance of three pre-trained models for generating arguments from a given context. Code completion is a software development process that significantly enhances developer productivity by automating the task of predicting code sequences as developers write code. Programmers use pre-trained models to reduce training time. Though using a pre-trained model makes a system faster, we checked the performance of argument generation accuracy. This empirical study also provides evidence on those expression types where the model failed to generate the argument.

Results from the study showed that argument usage context consists of better semantic information as it performed better than other models. Even when we added the structural information (in our study, expression type of tokens), it improved the performance of the model and provided a better understanding of how the context information affects the performance. Adding structural information helped to boost the performance-AUC value increases from 87.87% to 96.94%. The result from our second analysis showed that researchers should work with the poorly performed expression types.

1.3 Contributions of the Thesis

The contributions of the thesis are outlined as follows:

- Reimplement the DeepBugs approach for another dynamic programming language: Python.
- Evaluate the performance of DeepBugs for Python.
- Investigate the impact of the presence of method definitions paired with method calls. A noise injection technique is used to evaluate performance for all the method calls where the performance is reduced drastically.

- Develop a technique to detect swapping argument-related bugs for Python even if the method definitions are missing.
- Develop a technique to retrieve information on method definitions from the source code for their corresponding method calls. This provides additional information for code completion tasks and swapping argument-related bug detection.
- Investigate the importance of different context information sources for detecting swapping argument-related bugs.
- Conduct an empirical study using four pre-trained large language models for completing arguments from a given code context and analyze their performance.

1.4 Outline of the Thesis

- Chapter 2 describes prior studies related to this thesis.
- Chapter 3 explains the process of mapping method calls to their definitions.
- Chapter 4 provides a detailed description of name-based bug detection in Python.
- Chapter 5 describes an empirical study on the comparison of three language models for generating argument sequences for Python.
- Chapter 6 concludes the thesis by providing the significance of our work and outlining future research directions.

Chapter 2

Related Work

This chapter describes prior studies that support this thesis. This thesis is entirely based on the name-based analysis of the argument and its context. Therefore, this thesis is entirely on different methodologies or approaches that work with the relation between argument and parameter, detecting argument and parameter-related bugs, and the important features (i.e., Usage pattern-based context collection, mapping of method call and method definition), usage of Deep Learning Approach in software bug detection, and usage of large language models in software development. The following sections present a comprehensive summary of related works from which the problem statement, motivation, and proposed approach in the thesis engenders.

2.1 Name-Based Bug Detection for JavaScript

Pradel et al. [2] treated the source code in JavaScript as a bunch of natural language extracted method calls and their definitions from the same files and mapped them. Therefore, they built a bug detector using the line's local context, invoked the method call, and declared the line context of the method definition. They tried making a pattern using word2vec, where a simple neural network trains the correct and buggy code patterns to detect swapping argument-related bugs. They generalized the tokens by removing unnecessary tokens and replacing frequently used tokens. Though their model works perfectly for JavaScript, it may not perform similarly for other languages. Therefore, we first choose Python, the most popular language from the list of the TIOBE index [9]. Python is a dynamic language, and we used the same approach as DeepBugs to evaluate the performance of their model for another dynamic language. We found the accuracy of DeepBugs for Python was 61%. Therefore, the approach is structure and language-dependent. Consequently, we proposed our technique to overcome the issues of DeepBugs.

2.2 Name-Based Source Code Analysis and Bug Detectors

The name-based analysis [10] is a practical approach to considering the source code as a pool of information. A name-based analysis for equally typed arguments was described, where the detection of bugs is done by inducing semantic information from the code context. When it comes to equally typed arguments, there are issues in correctness with wrong orders and program maintainability for using unusual arguments in bad order. Another minor issue is poor understandability while using equally typed arguments in the same method call. These issues got extreme as the usage of equally typed arguments is not described in the code snippets. Their anomaly detection approach is a static analysis that gathers only the names extracted from a given JAVA or C programming language. This module extracted a set of arguments, which were name-based information. Therefore, a correct pattern of argument usage is gathered from the real word source examples, which implies that the position is fixed for that argument in the method call and definition pair. As it is an equally typed argument, the detector will differentiate between two arguments by gathering their name information from the line context. These names of arguments are converted to their unique numeric identities by the TFIDF approach. Though this method is almost accurate for static languages, it did not propose any information on how two arguments are equally typed if the type information needs to be included. On the other hand, it was not described if the method call has the same typed arguments and the same name. It leads to an end while this approach has an equal name and unknown type.

Programmers are providing information about a script in their comments and identifiers [4]. Their comments can easily help us understand the description of a program. DeepFix was [3] an effective method for detecting several bugs in a statically typed language where we need to train the model with a neural network and numerous faulty codes and correct codes. We are motivated by their approach in this angle that a pattern must be followed both in a buggy code and a correct code, which leads us to develop the wrong pattern from a code segment synthetically. A study by [4] was conducted on 100 computer scientists to give three names from a given code segment. They provided algorithms and snippets for their analysis. They created three variants of each function by changing the English entire word to a different name, a single-lettered name to a different letter, and the abbreviations to a common word. Therefore, when a programmer chooses a name for a variable, he must follow the categories mentioned above. Those three variants are not considered errors or bugs and project the significance of individual variables in a program. Though their study is based on JAVA, C, and C++, it is common for all static analysis events for dynamically typed languages. This approach is effective, while a self-supervised learning algorithm will learn from the code context and generate more data for recommendation. Patra et al. worked with the Name-Value Inconsistencies in Jupyter Notebooks in their study [6]. Their

proposed technique proved that the name of a variable is almost similar to its assigned value. They embedded 500k real-life name-value pairs from the Python source code and generated a pattern to detect Name-Value Inconsistencies. Therefore the assigned value and the line information can be used for determining a variable and its type which can be used for type information for any dynamic programming language like Python. Our model collected the assignment of those variables and generated embedding to detect their unique information from the source code. Another name-based learning approach [11] was implemented to detect bugs in JAVAScript projects. They described the problems of automatic bug detection models based on buggy and correct code examples. They collected information from code snippets to build correct and incorrect patterns for bug detectors of swapped arguments, wrong assignments, wrong binary operators, and wrong binary operands. These patterns are used as an input of an embedding generator to represent them in numeric format. Therefore, the unique representation of every token preserves the semantic representations. Their model worked well if data extraction was AST-based, which provided information related to the node and was preferable for performance boosting. To recover the meaning of variables that can be used as an argument, semantic information is gathered from mining the features from code. Though it is a static analysis, it provides meaningful features for bug detection. Therefore, using descriptive names or abbreviations is equally vital in their model.

The effect of name translation for general programs is effective, and while it is a user-defined code structure, it performs better. They converted the name-based approach [4] to an analysis-based approach. The name of a keyword or element in a program can be different from the actual name. Therefore, the structural relationship between the elements can provide accurate information for that detection. They analyzed five open-source applications in JAVA and collected all the relations for class, method calls, and inner classes. The relationship graph is introduced for all the instances of Figure Editors, MobileMedia, JFtp, JHotDraw, and Health Watcher and collects the data accidentally.

A way [12] to represent the source code to make information retrieval easier was introduced by reaching every entity in a source code, a novel approach to parsing the source code and generating an abstract syntax tree. Their approach is to detect the code clone and classify the source code by grabbing data from AST. They collected statistical knowledge and natural information from a code that captures lexical and statement-level features. This approach supports the detection of code clones and code classification. When they used a tree to connect the tokens from the abstract syntax tree, they captured most of the required data to detect the type of source code. Therefore, it is an effective idea to grab information from an AST to preserve lexical context.

Another promising approach to Java script programs to detect programming errors by a static analysis was performed by Berkay et al [13]. Their approach is to generate a template,

which is actually a textual representation of code, error, and error fixes. These error fixes are collected from the GitHub commits and their correct codes. To fine-tune the dataset, they manually analyzed a technique to gather correct and relevant bugs to reduce false positive results. As this model was developed with real-time bug examples and its fixes, it performs better than other static analyzers, although the bug fixes were not related to data flow in JavaScript.

A machine learning approach was trained from a large-scale, real-life buggy dataset. Miltiadis et al. algorithm was named BUGLAB [14], which has two models: the bug detector model with an additional bug repair tool and an approach to generate bugs from existing buggy files. They collected data from the code segments by an AST-based analysis to build a bug detector. Their model gathers the variable misuse, argument swapping, wrong operator, and wrong literals for Python programs. For their training model, they used a 3.4k PyPI dataset and 600 random PyPI test packages for evaluation. They used a GNN and GREAT transformer model to train the model [15] with real-world bug examples. By processing and feature extracting, they augmented the training examples and trained another similar model to reach new possibilities. This training model helps to generate warnings and detect bugs for their model. To add another aspect, they localized a bug from a code snippet. They concluded that repairing bugs rather than detecting them is hard. The reason behind this is anomalies in a program bug can detect the bug, and the possible repaired codes are endless. Therefore, understanding the context of the detected bug is mandatory to select the repair categories. This reduced their accuracy to 63

Detection of any keyword in a source code is challenging when they hold almost the same meaning but have different spellings—a technique [16] to collect the information for two different keywords with the same meaning. For example, programmers can use `getvalue()` and `getval()` in the same context. Though they are lexically different, they can be used in the same context or functionalities. Therefore, they proposed a technique to find the incorrectly matched identifiers by analyzing their lexical similarity and semantics information. Therefore, to collect data for their recommendation system, they gathered survey reports from the developers, filled in the code blanks from the programmers, and gathered a candidate list of similar parameters from the source code where they used the JavaScript files. This list of candidates is based on their context, the length of information, and word context from five lines. Their analysis shows that the model “IdBench.” performed well when the identifiers were from the same projects. Another aspect of their study is that none of the models performed well for argument similarity-based studies except those models based on training by the neural network. Though it does not perform well for dynamically typed languages, with its static analysis, the embedding can detect program anomalies and provides a new aspect to generate embedding for name-based analysis.

2.3 Parameter-Argument-Related Bug Detectors

The authors in [17] worked with the lexical similarity of the argument and parameter in Java programs. An argument with high similarity can be suggested in the same calling scope, which detects program anomalies and recommends correct arguments from a similar argument pool. They predicted that the parameters and arguments of the same name are related and follow the same properties. Their empirical studies showed how the names are similar to each other to build a pool of arguments with similar names. They collected the length of each parameter and argument to determine the length distribution of the arguments and parameters. After getting the name similarity and their length, their analysis shows that those pairs have less lexical similarity and fewer characters. It implies that the name was randomly declared, and there is no intention of using a similar parameter name while calling the method. Therefore, their experiment has information about all mapped parameters and their argument in JAVA real-world programs. Their hypothesis is entirely based on lexical similarity. A method call has two dependencies. Firstly, those features from a method definition of the corresponding method call, and secondly, the surrounding context of the location of the method call. Any discord of the method call may cause an anomaly, and while recommending the argument of a method call, this lexical similarity solves most of the challenges. When a method is called, the programmer will get suggestions of their arguments from the argument using their methodologies. pool. This recommendation for method argument is based on three cases. When a method call is inside the local scope, alternative arguments are only searched inside the similar argument pools from the same field access. If the argument is in a method call with a receiver object, the alternative arguments will be from similar arguments from the same object's scope. Their study shows that about half of the arguments do not have any alternatives. They proposed two applications of their approaches and found significantly good results. For anomaly detection tools in the Eclipse plug-in, the author chose a threshold to filter out the arguments from the similar argument pool to make the recommendation accurately. This will show a warning when the argument similarity is low and the difference between the current and potential sets exceeds the threshold.

In their second application of argument recommendation [17], the argument slot that must be filled is identified. A method definition will correspond to the method call, and they suggest similar arguments based on the method parameter. Secondly, considering the scope of a variable, the model will calculate candidate arguments and suggest an argument with the highest similarity values from the argument pool from the same field scope. Though the approach has higher accuracy, it has some restrictions in mining. Their analysis strictly showed that about half of the method arguments do not have any alternatives, which implies that if a method call or definition is entirely new, it will not recommend any argument or

detect any anomalies in the same project. On the other hand, the similarity values are calculated both using the method parameters and the argument. Therefore, it is mandatory to get a mapping of the method call with their method definitions and get information about the parameters. Thus, if there is no mapping, wrong mapping, and any missing library for a method call, it will ignore the anomalies and not recommend anything. Finally, it will recommend nothing for those method calls if they are different than the parameters, even though it is correct. Therefore, analyzing the code context and the usage of arguments for recommendations will add a new filter for the recommendation system. Another similar approach for detecting anomalies and code completion by parameter and argument name similarities is analyzed in [17] for JAVA and C programming. They collected the method parameters and arguments from a set of mapped method calls and definitions to calculate the lexical similarity. Their concept is to find the appearance of a similar argument in the peripheral of similar method parameters. The length of an argument is an essential feature to be matched with the corresponding parameters. It was shown that the parameter with a long length is prone to be much matched with the arguments with a long length. Although, sometimes, programmers use the abbreviated form of parameters and arguments, which will show more length inconsistencies and fewer similarities. This leads to a flaw in detecting anomalies and recommendation-by-name similarity-based analysis.

Selection [11] of the correct argument is always challenging for long codes as it is impossible to track down all values and types of variables easily. A large-scale analysis of the argument selection approach was studied to detect the defects in the correctness of argument selection. This paper describes the importance of argument selection defects and the result of wrong argument selection. Their approach started with finding the argument and parameter name similarities described in [17]. The argument is controlled by using identifiers, the scope [18] of those variables, the method's name, and the class constructors. They revised the history and found the argument and parameter-matching are inconsistent in several method calls and definition pairs. Under different threshold values, they verified the model, and their result showed that the ambiguity related to the argument name and parameter is solved by considering the field scope of their method argument. Though the method call and parameter are determined by using only name-based similarity, for similar code contexts where different variables are used for the same method, calls cannot be discriminated against and detected as an error.

The works mentioned above have related inconsistencies, solved by introducing program analysis and language models in a prior study [19]. This automatic recommendation of argument collects the syntactic type information and programming language-wise constraints to gather correct variables. Their main concern is detecting the arguments using the expression types and their induced values. They collected the data from GitHub filtering by stars, folks, and commits of repositories. Analyzing each project's syntax, they collected

the type and field accessibility. The collected arguments and the parameters are in the valid candidate list. Although all the arguments are candidates, they introduced a filter using a language model and parameter similarity. The approach is effective as JAVA is a static programming language where type information is collected by static analysis. It will not show similar output for dynamically typed languages, i.e., Python and Javascript.

Swapping two adjacent arguments [20] is a common mistake when programmers build large models. A static checker SWAPD was used to detect arguments swapping in C and C++ code. To generate the warning report, they introduced three checking systems (cover-based checker, sciatica vetting, and statistical checking). Similar to DeepBugs, they extracted names from C and C++ codes. The function names and argument positions are collected to develop the statistical database. The term morpheme was introduced to define the common terms in two arguments and parameters. The statistical database works on top of the cover-based checker and provides information about arguments at the call site. Therefore, a statistical checker will double-check the position and argument prone to be swapped. They have an analysis based on name generalization, and to balance the recall and precision, they have combined four stages of checking. Later on, for analyzing the code sequence, they used the same code analysis as DeepBugs. However, their process was based on the static typed language. Static bug detectors [21] are becoming popular because of their consistent data types and usage. Andrew Habib et al. studied an extended version of Defects4J Dataset on JAVA programs. The Defects4J dataset has real-world bugs, and the data is widely used in all the bug detectors worldwide. Finally, this dataset has the bug fixes with their corresponding bugs. They compared their model with Error Prone, Infer, and SpotBugs to evaluate their analysis. They used the code that has the bug and the after-bug-fixed code; the warning reported to the programmer before the bug led to an error for that code. Then, in the last step, they manually investigate the candidates generating bugs. Therefore, for a bug detector, they chose the candidates and differentiated between bug and warning messages. Their main intention was to identify how many of the 594 bugs could be detected perfectly by line-based, automatic, and manual validation. To detect the maximum number of bugs, users can use a combination of these three approaches, which reduces false positive bugs from the dataset. They worked on static bugs but did not provide any evidence of why the bugs were providing false positive results.

The authors worked with those method calls, which have multiple arguments but have the same types in another study [22]. Using the semantic information from the compilers, they extracted the meaning of identifiers to reduce argument selection defects. Though it performs well for C and C++, it will suffer from type-related ambiguities for dynamically typed languages.

2.4 Usage Pattern-Based Source Code Analysis

Usage patterns can be considered context to determine any token in a source code. A study [23] introduced a process that analyzed the context of a code and generated vectors from the usage pattern of the variable name. By capturing the semantic information from the tokens, the authors of that study added the meaning of a variable with their traditional vectors. Their approach performs well even though the variable names are poorly declared. From the large JAVAScript code from online, they generated vectors using a simple RNN model, which can be used for bug detection, malware detection, fixing syntactic errors, and code clone detection. They collected data from a large dataset to determine the name of their usage. This static analyzer tool is fully automated and provides the effectiveness of naming functions and variables from their usage context. They compared their work with JSNice and JSNaughty. They also analyzed whether the result was scalable or not. Using third-party libraries is quite common nowadays. Therefore, it is challenging to identify the right usage of a library in large software systems. Mohamed Aymen et al. [24] worked on the detection of libraries by analyzing the usage patterns of those libraries. Though they have collected usage patterns from the users or clients from the GitHub library, they found that most libraries have consistent usage patterns in most projects. After collecting the usage pattern from 6000 libraries from Github and Maven projects, they analyzed the maxEpsilon values, determining the pattern usage cohesion metrics. Therefore, the mined pattern from all the libraries with a threshold value provides a better system to detect the correct library. Though these libraries change dynamically daily, the author considered a universal structure that will be constant for every system. This approach effectively builds usage patterns related to detection tools and code completion contexts. Another study was conducted to increase the semantic information from the code and support code suggestions by mining the usage context, a n-gram topic model with its associated elements from the same code. From the semantic information, anyone can understand the exact meaning of the code. Therefore, it is effective for a bug detector to identify the pattern of elements for code completion and source code repair. They collected semantic code tokens directly associated with an ID, roles, data type, and especially data dependencies. In addition, they collected the functionality of the token for the token. On the other hand, to preserve a variable's code context and meaning, they collected all the semantic information from the code for a particular token by considering the variable's scope or token. This scope will show how these elements are used in their particular location only. Therefore, this local context from the same file showed that the generated vector contains unique information, which showed higher accuracy in real-life projects while working with tokens.

We are highly motivated by the works mentioned above and formulated our work based on name-based analysis, bug detectors, swapping argument-related bug detection, natural

language processing for source code analysis, and finally, context collection based on usage patterns. We treated any source code in Python as a source of natural information engendered by programmers. Therefore, first, we consider that the name of an identifier or an argument contains valuable information for representing a source code. We preferred to analyze and detect the bugs by accidentally swapping two adjacent arguments. Previous state-of-the-art work with JavaScript and proposed a technique to generate data from existing source code synthetically. Our approach worked with many examples with all the Python method calls. In contrast, their approach was limited to considering method calls only from the same file with accurately mapped method calls. When we collected context from the lines, an argument was found; we could not grab enough information from the local context. Then, we brought up the usage context of the argument as a determining factor. Our contribution is that without knowing the information of a method declaration for a corresponding method call, how can we detect this swapping argument-related bugs for a method call? We found that an argument is determined from the before lines of their invoked lines, and we collected them to train our model with the usage pattern. As Python is a highly dynamically typed language, we have collected the expression type of an argument for training the model with the information that this method took an expression previously; now, it might take a similar pattern or similar expressions for the same method call. Another pattern-based study for generating the subsequent APIs was introduced in [25]. They collected all the related APIs from a source code and generated the graph-based occurrence flow for all of them. Mining the usage case scenarios and the specifications of an API provided the model information about which API can be called after which API. These use cases drove the model to generate frequently used APIs. They used a model checker, which consist of data-flow-intensive. When mining APIs, they subtly collected that API usage sequence where the support value(determinator of choice of traces of API sequence) is higher. Though their model is entirely based on static source code analysis, it shows how the usage case scenario helps the API recommendation. Parameters contain promising information, which was used in another argument recommendation system [26]. Their approach was to build a generalized database of argument usage patterns. Therefore, they collected a one-to-one mapping of parameter usages for a method call. As the approach is based on JAVA source code, it follows a one-to-one variable and types information data. This information reduces the ambiguity of argument selection. Still, it is a prominent idea to use the context that appears for a method argument and method call, which will be almost the same for a similar method call. As mentioned above, these studies worked with API calls and their sequence. In contrast, another study [27] [28] [29] focused on all the variables by running the program and recording the changes in their values. Daikon’s output is used for the generation of test cases, prediction of incompatibilities in component integration, automation of theorem proving, repair of inconsistent data structures, and validation of

data stream integrity based on the variables' usage from a source code.

2.5 Deep Learning for Bug Detection

In a project, a bug can be generated at anytime for several reasons, such as data inconsistency, wrong usage of variables, improper control statements, etc. However, it is an ideal approach to study the generated errors by solving them manually with human interaction, which has the issue of time complexity and is strenuous. Therefore, deep-learning techniques are used to build automated models from the source code. The name-based analysis [10] [4] [11] gave us leverage to treat the source code as a natural language example. Therefore, this information from the source code can be used by extracting data token-wise, and by using machine learning and deep learning or pre-trained models, an automated bug detector can be built. A prior study [30] introduced how a deep learning model can detect bugs from static source code information. The surrounding context of a token was collected, and by word2vec, they transferred the code context to a vector space. This transformed numeric data was used for a deep-learning model that can be used for bug detection. Besides, the tokens of a source code need to follow a certain sequence. Changing the sequence of these tokens will generate bugs or warnings during execution. Therefore, this study was designed to collect static information from a source code and generate a warning when it finds the wrong sequence of tokens in that code snippet. The study showed promising results for Swapped arguments, Wrong assignments, Wrong binary operators, and Wrong binary operands. It implies that name-based context information can detect bugs or warnings by a static analysis. Again, the localization of bugs by information retrieval was described in a study [31]. The static information from the source code and the textual similarity with this infraction and bug report were collected as a feature to train a deep learning model. Therefore, code retrieval is used to extract the feature from a source code, which is used to detect bugs.

Another author, in their work [32], proposed a technique to tokenize the source code and collect the required tokens as a feature. A Recurrent Neural Network (RNN) converts these tokens into vectors. These vectors are called embeddings. When they represent the source code as AST, CFG representation, it is easier to access the required fragments from the source code. Therefore, these representations are used to normalize the source code and to feed deep-learning models.

These works are related to inconsistency and anomaly detection of method calls and their arguments. Therefore, we proposed a technique based on the abovementioned approaches and proved that a usage pattern of an argument as a variable is adequate for detecting swapping argument-related bugs in Python. Our proposed technique performs well even if the method definition is unavailable. On the other hand, our technique is good enough for

any large-scale studies.

After detecting anomalies regarding the method call and its argument, we are skeptical about how to generate the arguments of a method call. For dynamically typed languages, most of the compilers (i.g. Visual Studio Code, PyCharm, Jupyter Notebooks) follow the pre-trained models and processing for suggesting code segments (especially- API suggestion API argument competition). Therefore, we intend to investigate the accuracy of large-language models and check manually whether these models can generate the argument list ideally or not.

2.6 Usage of Large Language Models

In a recent study [33], the authors collected the partial codes from Java and all the expressions used in the arguments [34]. By filtering the argument and the scope of variables, their model refined the list of possible arguments and generated a list of candidates. This model takes an input sequence of partial code, and the method the requested position of the argument and generates the input query for an argument request. Their approach outperformed the GPT-2, CODET5, and SLP. Our analysis checked whether the large language model performs the same for the Python method calls. Code completion should be more accurate when the used model has enough information to detect the pattern. This argument recommendation is also a specific type of code completion. In their study, another author used the large language model to check the performance of generating code chunks from a given partial code [35]. Transformer-based neural architecture was used to fine-tune the model and create a segment for a given method call. This method of body completion is a token-level completion, which performed well for Java, a statically typed language. Therefore, large language models can generate tokens for dynamically typed languages. For predicting the subsequent tokens [36] from the given code context, a study [35] conducted a large-scale study comparing the performance of Deep Learning models and Large language models, for example- BERT. The generated tokens were compared with the original tokens by considering the BLEU score and Levenshtein distance. They conducted this analysis for an Android dataset [37] and a JAVA dataset. They compared the RoBERTa model with the typical n -gram model and found that the RoBERTa generated better results for both single and multi-tokens. Therefore, we are highly motivated to check the performance of RoBERTa for the Python dataset. It was challenging to generate the whole line of code sequence for Python. A study [38] was conducted to create an entire line from a given context by neural language models. Their findings indicate that Transformer language models consistently outperform RNN-based models across both datasets, aligning with prior research in language modeling. The whole line was generated using GRU and Transformer (GPT and GPT-2) [39]. They collected tokens from code segments and Syntax-based con-

texts from AST, which are used as input for the GRU+Token model, GRU+Syntax model, TransformerLM+Token model, TransformerLM+Syntax model, and according to their evaluation, Transformer language models are generating better results than the others. The model must have the tokens to generate tokens from a given code context. It is impossible to generate a name of an argument that is not even in the given context. This study [40] [41] was designed to generate tokens using an LSTM model but providing all the local and global variables candidates for a requested API. It is mandatory to get the argument prediction; the argument must be in the context, and a rule-based data flow of variables is needed to collect the context from the source code. Besides, a study introduced three types of query procedures of language models. Left-to-right language Models, Masked Language Models, and Encoder-decoder Models are used to generate tokens. As they are pre-trained models, the authors have used them and built another model, PolyCoder, to develop on top of GPT-2. Thus, large language models can be used Marcel et al. [42] described how context or tokens are used for a code recommendation system. They proposed a technique to improve the code completion system by collecting the frequency of the tokens, using the association mining rule to collect those tokens that appear together. As their models used these tokens for code completion, we have collected those contexts for our large language models, which are highly frequent and associated with method argument. In a code-completion study, Matteo et al. [43] used the RoBERTa model for training and generating code tokens. They used masked tokens to generate code tokens, the BLEU score, and Levenshtein distance to evaluate the predictions. Their study showed that for Python programming language, their performance is higher than that of other n-gram models. Therefore, we used the RoBERTa model for token generation and compared the generated tokens by calculating the BLEU score and Levenshtein distance with developed arguments and original arguments.

2.7 Conclusion

Our thesis is to analyze the method calls and their arguments in Python. Most programmers need to be made aware of an argument's datatypes and expression types. Therefore, we proposed our technique to verify the variables from the source code context and generate a warning if there is a swapping argument-related issue. Though retrieving the method definition is easy, it is challenging to map the method call to its actual definition. Our approach also collected the method definition and mapped to its method call. This increases the domain of our analysis and provides the model with extra information. Our proposed technique dynamically collects the usage information of variables from the source code. It proves that usage pattern study is one of the promising approaches to building a bug detector and anomaly detection. Our study was used to support our later chapters on argument generation from code context and calculate the performance of a large language

model for argument generation in Python by analyzing the scoped variable lists from the source code. We also proposed a technique to verify where the large language models fail. To the best of our knowledge, this is the first context-based bug detection and empirical study of argument in Python. Pradel implemented this name-based learning approach to detect bugs in JAVAScript projects. They described the problems of automatic bug detection models based on buggy and correct code examples. They collected information from code snippets to build correct and incorrect patterns for bug detectors of swapped arguments, wrong assignments, wrong binary operators, and wrong binary operands. These patterns are used as an input of an embedding generator to represent them in numeric format. Therefore, the unique representation of every token preserves the semantic representations. Their model worked well if there was data extraction with AST-based, which provided information related to the node and was preferable for performance boosting. To recover the meaning of variables that can be used as an argument, semantic information is gathered from mining the features from code. Though it is a static analysis, it provides meaningful features for bug detection. Therefore, using descriptive names or abbreviations is given equal importance in their model.

Chapter 3

Linking Method Calls to their Definitions

3.1 Introduction

Programmers declare variables and use functions very repeatedly. The correctness of a program generally depends on the assignment of the variable type and passing a correct typed variable as an argument to a method call. Therefore, we analyzed the induced type of a variable and proposed a technique to detect the type from the source code context. While investigating the type from an assignment operation, we found programmers are assigning a method call as a value for a variable, and this method call points the definition to get the returned value. Therefore, we proposed another approach that will bring the method definition to its method call.

3.2 Dynamic Type Solving of Variables in Python

We are working on dynamically typed language. In Python, we declare a variable without declaring its type. This type is assigned according to the data provided to the variable at runtime. For example, in Figure 3.1, in the following code snippets, we have a variable `METADATA_FILE`, defined at line no. 3, using the assignment operation. To detect the variable type, we need to analyze the value assigned to the variable with its local field access. The variable can get values anytime in the program and can be changed anytime, anywhere in the code script. In the code snippet at segment 1 in Figure 3.1, the variable `METADATA_FILE` got a value and was used in the function call “`io.open`”. Therefore, the method call expects the string type or the path type as the first argument and the string type as the second argument(here, it is ‘`r`’). In the code snippet in segment 2 at Figure 3.1, the variable `METADATA_FILE` is assigned three times and used in the function at

```

Type_example.py • Type_example_2.py 4 •
F:\data> 13 > eliviorverdier > dispatch > dispatch > Type_example.py > ...
1 import io, json
2 #segment 1
3 METADATA_FILE = "zonefile_metadata.json"
4 with io.open(METADATA_FILE, 'r') as f:
5     |   | metadata = json.load(f)
6
7 #segment 2
8 METADATA_FILE = 3 # assigned an integer value
9 METADATA_FILE = metadata.split("\n") # assigned a value which is returned by metadata.split("\n")
10 METADATA_FILE = "zonefile_metadata.json" # assigned a value which a string or a path type value
11 with io.open(METADATA_FILE, 'r') as f: # expects the 1st argument as path type & second arg as string
12     |   | metadata = json.load(f)
13
14 #segment 3
15 METADATA_FILE = "zonefile_metadata.json"
16 with io.open(METADATA_FILE, 'r') as f:
17     |   | metadata = json.load(f)
18
19 #segment 4
20 def method_metadata_fetch(METADATA_FILE):
21     data_path=METADATA_FILE
22     with io.open(data_path , 'r') as f:
23         |   | metadata = json.load(f)
24 method_metadata_fetch("zonefile_metadata.json")

```

Figure 3.1: Example of Dynamic Type

line no. 11. Therefore, the variable was changed to 3 types. In the example in Figure 3.1, when the programmers analyze the code snippets, they will understand the code pattern and the assignments of METADATA_FILE. In line 8 at segment 1 in Figure 3.1, the value is assigned to the variable as an integer, and in the next line(9), it immediately changes to a different expression (a method call). As this expression is a method call, the value for METADATA_FILE will be the method call’s return value. Therefore, the final value from the usage pattern is the latest usage of the argument’s assigned value. We divided our approach into three steps. First, we have extracted all the assignment variables. We have also differentiated the variables based on the local and global variables. In segment 3 in Figure 3.1, the METADATA_FILE is a variable in the global scope of the variables and has a method definition and has a parameter METADATA_FILE, which has scope under the method definition “method_metadata_fetch”. We have determined the scope of the variables and differentiated them to generate the argument usage pattern accurately. On the other hand, in segment 4 in Figure 3.1, “data_path” is a local variable with scope only from line no. 20 to line no 23. Another example is in segment 1 in Figure 3.2, where a value is assigned to Sum, where it is the Binary operation. In the second line, Student_1.marks is a returned value from “get_marks” and can be any type.

While generating the argument recent usage pattern and determining the argument type, we found 24 expression types (see Table 3.3) frequently used in Python. However, these expressions carry different values, which are the type-defining factors. We divided our type information into two categories. They are basic types, secondary types, or inferred types.

```

Type_example.py • Type_example_2.py 4 •
F: > data > 13 > olivierverdiere > dispatch > dispatch > Type_example_2.py > ...
1 #segment 1
2 Sum= 5+6
3 marks= 78
4 Student_1_marks= get_marks(subject,marks)
5 def top_result ( Sum_value, marks):
6 |     return Sum_value - marks
7 result_student_1= top_result ( Sum, Student_1_marks)
8
9 #segment 2
10 name = None
11 axis_list = [1,2]
12 axis_list_processed = [1,2]+ other_arg
13 scope = whoosh.fields.STORED
14 #segment 3
15 def percentage_calculator(data, total):
16 |     return data/total*100
17 student_1_marks=percentage_calculator(80,700)
18

```

Figure 3.2: Example of Dynamic Type(contd.)

- Basic Types:** This type of information is visible and can be extracted by analyzing the source code. These are also known as static types. These variables are independent, and the inferred types are dependent on these independent types. We found that 11 expressions exactly followed the data type of the variable. String, List, Integer, Dictionary, boolean, None, Tuple, float, Lambda, bytes, and Set(see Table 3.3) are treated as basic types in Python programming language. In Figure 3.1, at line no 10, the variable “METADATA.FILE” has taken the string as a value of the variable. For example, a string must be enclosed with an inverted comma. If it is a byte type, it must start with “b”. However, this approach is extended and used to detect the type of binary operation. In that case, generally, we traverse the whole value of the assignment to match any pattern(String, List, Integer, Dictionary, boolean, None, Tuple, float, Lambda, bytes, Set) that is available or not. If any of the patterns are matched, we directly assign the type of the matched pattern to the assigned variable or keep it unknown.
- Secondary Types:** Some of the expression types - Attribute, Call, BinOp, Starred, Compare, UnaryOp, Lambda(see Table 3.3) - are considered Secondary types. Due to the complex expression types and the internal operations in those expressions of variables, the type of those assignments depends on those operations. For example, we provided examples of their solving process in the table.

We collected the information as follows in Table 3.2. For a collection of this type, we generated a knowledge base, which will be updated every phase with new examples. The process is as follows-

Table 3.1: Description of Updating the Knowledge Base for Type Detection

Knowledge Base	Description
(BT)	Database with only Basic Types
(BT+NT)	Database with Basic Types and Name Types
(BT+NT+AT)	Database with Basic Types, Name Types and Attribute Types
(BT+NT+AT+ST)	Database with Basic Types, Name Types Attribute Types and Subscript Types
(BT+NT+AT+ST+CT)	Database with Basic Types, Name Types Attribute Types Subscript Types and Call Types

1. First, we made a database(BT)3.1 with all assignments with the basic type on their right side as the value of the assignment.
2. We took all the assignments with the “Name” type expressions as values. After extraction, we checked the right-side names to the left side of the assignment database, where we saved the basic types. If we found a direct match, we added them to the previous database(BT+NT)3.1.
3. Then we picked all the all the assignments with the “Attribute” type expressions as values. Then, we checked the class instances and the updated database (BT+NT+AT)3.1.
4. We picked all the all the assignments with the “Subscript” type expressions as values. Then, we checked the usage of the name of the subscript and the updated database (BT+NT+AT+ST)3.1.
5. Now, we used the mapping module to find out the method definitions and their mapped method calls. We also collected the segment of a method definition block and tracked it down from top to bottom to get the actual value of a return block of a method definition. We used the technique of automatic return type detection ¹ to gather all the return types of the method definitions. Then, we filtered out all the method calls used as a value of the assignment. Typically, these method calls were already mapped by our algorithm described in the section 3.3.1 and collected the type of a method definition by following the process from ². Thus, we gathered the type of maximum number of mapped method calls from examples. Then, we updated the database as (BT+NT+AT+ST+CT)3.1.
6. Therefore, our database (BT+NT+AT+ST+CT)3.1 gained the possible types of 60% of the assignment from all project examples. Then, we used the database to solve

¹https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4088424

²https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4088424

the binary operations as it combines multiple expression types. The combination of expression types can be any of the type expressions with the binary operation. We simplified the process by two steps from the equation 3.1.

$$Type_of_BinOP = \begin{cases} Basic_Type, & \text{if Basic_type with other_Expression} \\ Check \\ Dataset \\ forType \\ oftheExpression, & \text{Any Expression with Any Expression.} \end{cases} \quad (3.1)$$

Table 3.2: Type Collection from Figure 3.2

Segment no from Figure 3.2	Variable name	Assigned value	Assigned value (Expression Type)	Probable value Type
1	Sum	5+6	Binary operation	int
	Student 1 marks	get_marks (subject, marks)	Method Call	Have to find the return value of the method call the get marks
2	name	None	NoneType	NoneType
	axis_list	[1,2]	List	List
	axis_list_porcessed	[1,2]+ other_arg	List + Name	(must be a list)
	scope	whoosh.fields.STORED	Attribute	Depending on the sourcecode

3.3 Linking Method Calls to their Definitions

This section describes the process of linking method calls to their definitions. This is required to support the implementation of DeepBugs, a state-of-the-art name-based bug detection technique. DeepBugs uses a semantic vector representation of method arguments or method parameters. We implement DeepBugs for the Python programming languages to compare with our proposed technique.

Significance :

- This approach will map the method call to its actual definition to generate a sequence

Table 3.3: Classification of Expression Type Extraction

Variable Type	Variable Name	Assigned value (Real)	Assigned value (Expression Type)	Detected Type
Primary Types	i	1	i=1	integer
	st_name	“Harry”	st_name=“Harry”	String
	greeting	b“Hello, world!”	greeting= b“Hello, world!”	byte
	arr_val	arr_c[3]	arr_val=arr_c[3]	Subscript check code lines
	ip_seq	(1, 2)	ip_seq=(1, 2)	Tuple
	trigger	True	trigger=True	Dictionary
Secondary Types	schema	whoosh.fields. STORED	schema= whoosh.fields.STORED	Check class or objects from code lines
	type_data	msg	type_data= msg	Name Expr check code lines
	st_marks=	percentage_ calculator(80,700)	st_marks= percentage_calculator(80,700)	constant depending on return type of method call
	content_string	contents % unit	content_string = contents % unit	depending on the left and right part of binary operator

of parameters and arguments.

- This approach will support solving the type of those assigned values, which are called expression types.

3.3.1 Method Definition Pattern

Method definitions can be divided into two categories. It can be user-defined, which is project-specific and declared by users, and the pre-defined or library functions used by importing the libraries in a file. When we call a method, the IDE maps the method call with its corresponding method definitions to use the arguments that can be passed to the method definition as parameters. Therefore, it is challenging to map the method call with the definition of the corresponding method.

Besides, the method definition can be declared in the same file or different files in the same or another project. By the “import” keyword, we enhance the scope of a method definition



Figure 3.3: Different Types of Method Calls

(from another file) to the imported file. Our thesis proposed a rule-based heuristic approach to link the method definitions (defined in the same file and the same project) with their method calls. It implies that the method declaration can be in the same file where the method is called in the different files in the same library. We aim to map the definitions defined in a project locally with their corresponding method calls and build the correct binder for the correct code pattern. We parsed the source code for each project, and our code extractor module generated two files (Collection of method calls and collection of method definition). We have another two modules that will collect the class information and collect “connected files” of any files given a file path by its import statements. We categorized the calls based on the location of the definition and method call. It implies that the method declaration can be in the same file where the method is called in the different files in the same library.

3.3.2 Method Call Patterns

In Figure 3.4, a very method called “lxml.html. from string (page)” is shown in line 141. The purpose of the method call is to parse the HTML content provided in the “page” variable and return an Element object that serves as the root of the HTML element tree. This tree can then be navigated and manipulated using various methods provided by the lxml.html module. If we carefully look into the method call, we found the patten of a method call can be a sequential call of python.library_name.submodule_name(or class_name).function_name. This implies that the search string for a method definition for a method call will use the sequence of the method call followed by the import statement. Therefore, each call must be related to:

- Only the method call sequence(Same File Method Call Mapping)
- Both an import statement and method call sequence(Outside the same file Method Call Mapping)

Our proposed heuristic searching algorithm goes through both concepts and brings the method definition with its call.

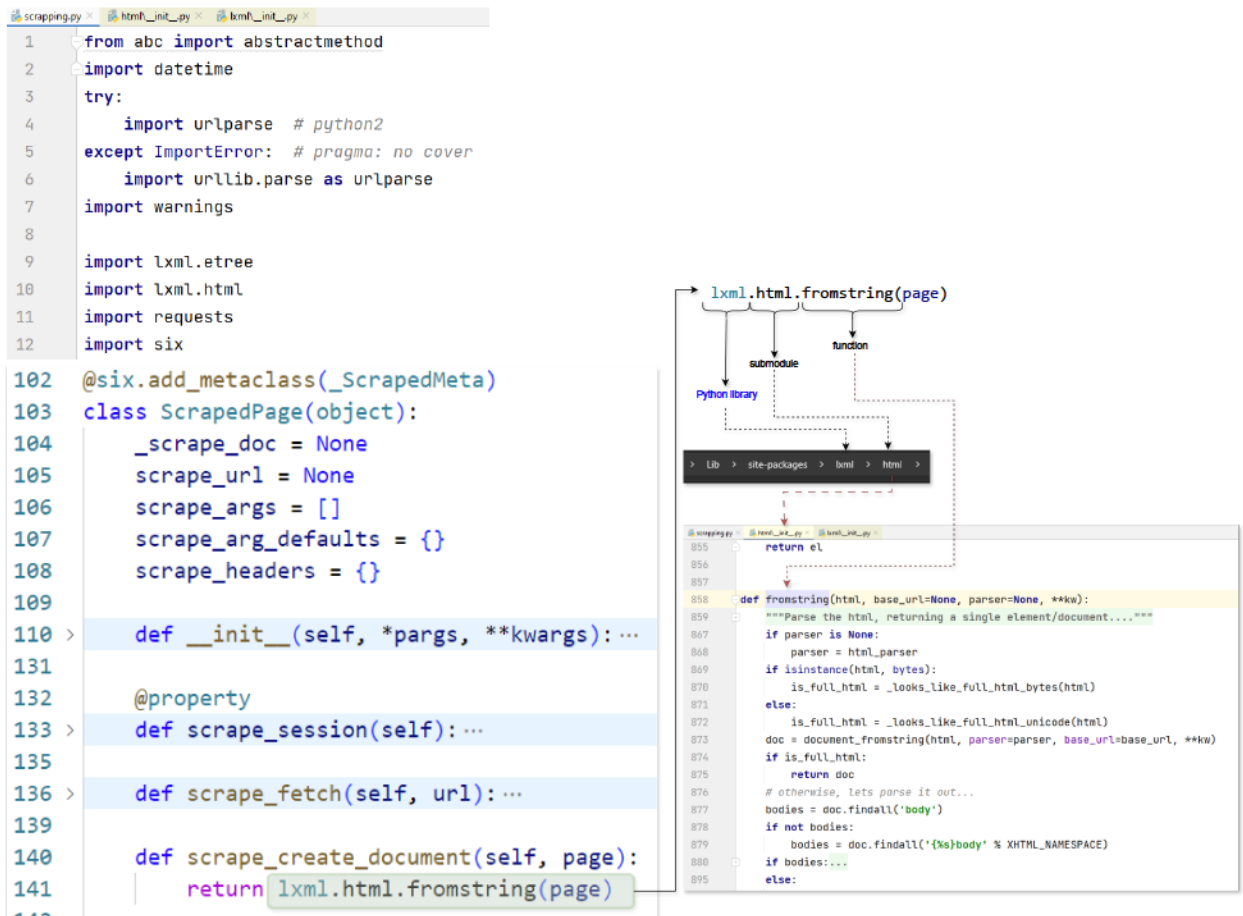


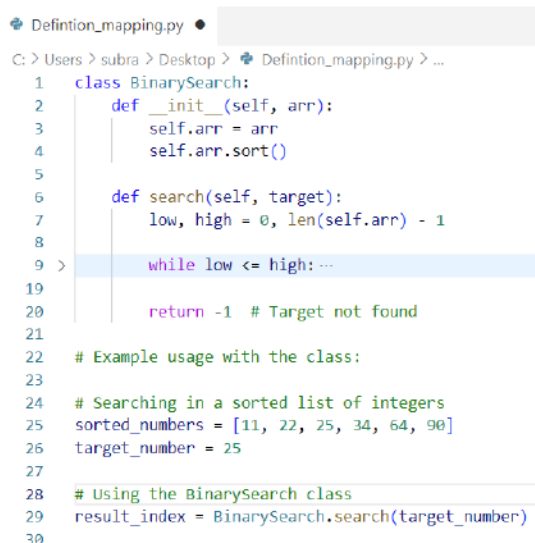
Figure 3.4: Method Call Pattern

3.3.3 Analysis 1: Method Call and Method Definitions in the Same File

We aim to map the definitions defined in a project locally with their corresponding method calls and build the correct binder for the correct code patterns. The method definition can be declared directly in a module or class. First, the method calls and definitions were collected from a file with their parameters and arguments. Then, we matched the name of the method call to the name of the method definition. If the method definition is defined in a class, then we match the name of the class name with the attribute name.

A method call can be called from a function definition, a class method, or an inheritance of a class. In Figure 3.3, the first segment shows the function call and its declaration. We denoted it as a direct function call. The second segment projected a class declaration at line no. 1, which has a method declaration “calculate_age.” To access the method from

the class “Student,” we have to create an object of “Student” or use it directly by calling “Student.calculate_age()”. The third segment of Figure 3.3 showed the class inheritance procedure where a class “LoginView” inherited the instances from another class “RedirectURLMixin,” which implied that for the case scenario of inheritances, we have to look up the inherited class for determining the scope of a method declaration. If a method definition is located inside a class, we also collected the name of the associated class. Given a method call without any receiver expression, we looked for a method definition outside classes in



```

Defintion_mapping.py
C: > Users > subra > Desktop > Defintion_mapping.py > ...
1 class BinarySearch:
2     def __init__(self, arr):
3         self.arr = arr
4         self.arr.sort()
5
6     def search(self, target):
7         low, high = 0, len(self.arr) - 1
8
9         while low <= high: ...
19
20         return -1 # Target not found
21
22 # Example usage with the class:
23
24 # Searching in a sorted list of integers
25 sorted_numbers = [11, 22, 25, 34, 64, 90]
26 target_number = 25
27
28 # Using the BinarySearch class
29 result_index = BinarySearch.search(target_number)
30

```

Figure 3.5: Method call and its Corresponding Definition in the Same file

the same file by comparing their name and number of parameters. If the method call has a receiver expression, we determine whether the receiver expression matches the class name located in the same file. If we find an exact match, we look for a method definition located in the class whose name and number of parameters match the name and number of arguments of the method call.

In this Figure 3.5, line 1 contains the class declaration “BinarySearch”, and line 6 contains the method declaration “search”. A method call was found at line 20, “BinarySearch.search()” in the same file. We first mapped with the file name and then checked if any instance was used in the method call. This brought up the method definition and method call for the current file. After matching the file names, we perform a heuristic mapping with the name of the method definition and the name of the method calls. We have collected all the name-matched method definitions for that method call. This mapping generated one method call mapped with one or many method definitions. We re-mapped the mapped method calls and their definitions to remove this dilemma by matching the attribute name of a method call with the class name of their corresponding method definitions. As an example from the Figure 3.5, the first “search” keyword is matched directly

with the definition in the same file and then compares the attribute of “search” (in this case, “BinarySearch” is the method attribute name) with the class name of “search” (in this case “BinarySearch” is the class name). Therefore, we collected mapping data in the same file. We found that method calls and their mappings are only 5-7% of the total method calls. The definitions of the unmapped method call are spread in different files and libraries. We studied and retrieved the method definitions from different files in the same project.

If the receiver of a method call is “self”, the method definition and the method call are located inside the same class scope. For example, consider the figure3.6, method call: self.bool(). The method bool() and the corresponding definition must be located inside

```

1 class Stack:
2     def __init__(self) -> None:
3         self.items: list[Any] = []
4
5     def push(self, item: Any) -> Self:
6         self.items.append(item)
7         return self
8
9     def pop(self) -> Any:
10        if self.__bool__():
11            return self.items.pop()
12        else:
13            raise ValueError("Stack is empty")
14
15        def __bool__(self) -> bool:
16            return len(self.items) > 0

```

Figure 3.6: Example of self as a Receiver of A Method Call

the same class or any inherited class of the same class. We establish the mapping by searching class methods in the current scope. If the heuristic approach of searching method declaration fails, we further checked the inheritance of the class for class method declaration. Our approach is based on the fact that the inherited class must be treated in the same scope as the object of the child class. Therefore, in the third segment of Figure 3.3, any object or instance of class “LoginView” must have the right to use the methods of class “RedirectURLMixin”.

To resolve this class inheritance, we have first checked the same class for the method call and the superclasses in the declaration of a class method. This extends the scope of a class. In the third segment of Figure 3.3, The class “LoginView” is inherited from the class “RedirectURLMixin.” Therefore, for mapping of the method, call “get_success_url” at line no. 81 will first check the method definition inside the class “LoginView” and cannot find the definition, and then it directly searches the inherited class “RedirectURLMixin” and “FormView”. At line no. 40, the class “RedirectURLMixin” has the class method “get_success_url,” which was mapped to the method call at line no. 81.

3.3.4 Analysis 2: Both an import statement and method call sequence (Outside the same file Method Call Mapping)

If the method definition is not present in the current file, we import them from other files in Python. These imports can be files, classes, a directory, or a particular method. Therefore, when a file includes any import statement, it imports those modules, files, and the class, and they must be considered a part of the same file even though they are from different locations. When we get all the files connected to a file, we follow the same approach described at ??.

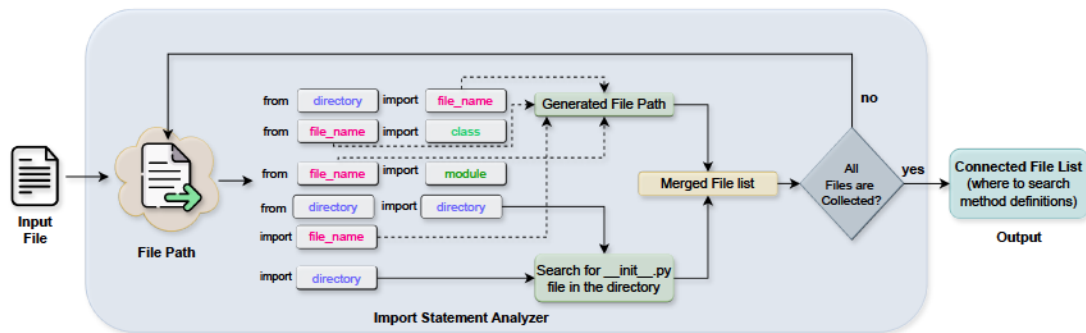


Figure 3.7: Import Statement Analyzer

First, we parsed a file, collected all the import statements, and gathered all the files related to the import statements. If the files are in the scope of the project, we mention them as connected files. Another major feature of Python is that Python has the “`__init__.py`”. When in a Python script there is an import statement pointing to a project’s directory, first

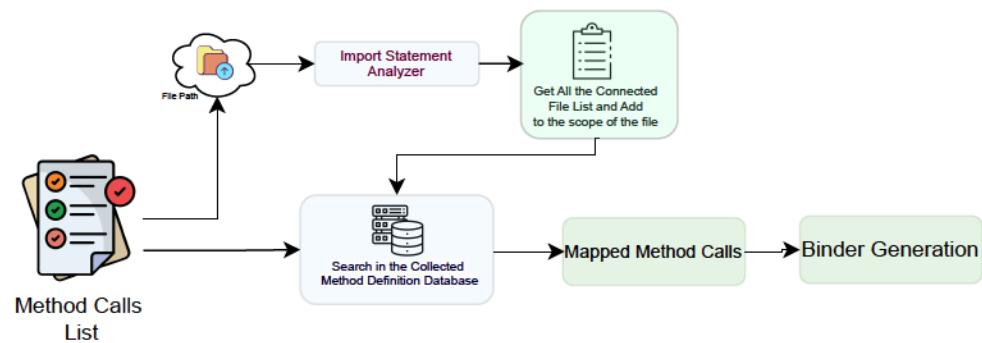


Figure 3.8: Usage of Import Statement Analyzer to Generate Binder for DeepBugs

we have to search the existence of the directory and secondly, in the directory, the existence of the “`__init__.py`”. In Figure 3.7, the module Import Statement Analyzer took a file as an input and generated a list of connected files. This Analyzer was used to generate binders for the DeepBugs model in Python (3.8). If the “`__init__.py`” file is available, we need to parse the file collect the related Imports, and add them to all the files in the same

directory. It is a functionality or workflow related to all the files in a certain file directory. After collecting the files connected with the particular files, we treated them as a part of the same file and checked the module name and then the class name by following the ?? and collected the mapped method calls. As in Python, we found more than 60%-70% of method calls are library imports, which we have mentioned as global imports, and the rest of the 30%-40% of method calls are user-defined or local method calls. We got the correct mapping of 80% of method calls out of all the local method calls, which are our correct code examples. The binder generation process from the method call is described in the following section.

3.4 Example of Resolving Method Call

We solved our collected method calls by using our above-mentioned approach in section 3.3.3 and section 3.3.4.

Method Call Mapping Rule-1: In the first segment of Figure 3.3, there are 3 method calls at lines 5, 8, and 8. The methods call “print” and “factorial” are used. For “factorial” we checked the file for method definition. As “factorial” does not have any Python library or any sub-module with its block, it is a direct mapping to its method definition. Therefore, we followed the steps-

- Match the outside class method definitions by their names and make a list of matched method lists.
- Check the number of arguments of method calls and the number of parameters of the matched definition.
- check the number of default variables for method definitions and ignore the list of default variables for method call as it is passed automatically when a method is called.

Result: We got a one-to-one mapping with “factorial” at line 8 to the definition at line 1 3.3

Method Call Mapping Rule-2: In Figure 3.5, we found a call at line 29. According to the pattern of method call at 3.4, the method call “BinarySearch.search(target_number)” has two parts. The first part, “BinarySearch,” is the class or sub-module, and the second, “search,” is the function. Therefore, we follow the following steps-

- Match the inside class method definitions by their names and list the matched method lists with their class name. If it is an exact match, then get the list.
- Check the number of arguments of method calls and the number of parameters of the matched definitions.

```

1   from abc import abstractmethod
2   import datetime

191  class CssDate(Css):
192      def __init__(self, selector, date_format, **kwargs):
193          self.date_format = date_format
194          super(CssDate, self).__init__(selector, **kwargs)
195
196  def cleanup(self, value, elements, scraped_page=None):
197      try:
198          return datetime.datetime.strptime(value, self.date_format)
199      except ValueError:
200          return None

```

```

1563  * class datetime(date):
1564      """datetime(year, month, day[, hour[, minute[, second[, microsecond[, tzinfo]]]]])"""
1569      __slots__ = date.__slots__ + time.__slots__
1570
1571  * def __new__(cls, year=None, month=None, day=None, hour=0, minute=0, second=0,
1572              microsecond=0, tzinfo=None, *, fold=0):...
1605
1740
1947  @classmethod
1948  * def strptime(cls, date_string, format):
1949      'string, format -> new datetime parsed from a string (like time.strptime()).'
1950      import _strptime
1951      return _strptime._strptime_datetime(cls, date_string, format)

```

Figure 3.9: Example of Mapping a method call to method definition from another file

- check the number of default variables for method definitions and ignore the list of default variables for method call as it is passed automatically when a method is called.

Result: We got a one-to-one mapping with “search” at line 6 to the definition at line 1 3.5 from the class at line no 1.

Method Call Mapping- Rule-3: In Figure 3.4, we have a method call at line 141, and there is no definition or class method that follows the call pattern “ lxml.html. from string (page)”. Therefore, it is necessary to search through the import statement where we found “import lxml.html” and when we used our import statement analyzer from 3.3.4, we found a set of list of connected files. As “import lxml.html” is a directory, it has brought the “_init_.py” file from the directory lxml.html and parsed the file. Therefore, it came under the scope of the current file “scraping.py”. Then, we follow the same approach to match the method call name with its definitions.

Result: we found the method call “`lxml.html.fromstring(page)`” is mapped with the definition in the file “`__init__.py`” in the directory “`\\Lib\\site-packages\\lxml\\html`”.

Method Call Mapping- Rule-4: In Figure 3.9, at line 198 a method call is found “`datetime.datetime.strptime(value, self.date_format)`” . As the method definition was not found in the same file, we used the import statement analyzer tool to find the imported file list and found that `datetime.py` is in the scope of the current file.

Result: We followed the method call pattern and found the “`datetime`” as a library. In the library, the `datetime.py` file contains the class “`datetime`” where “`strptime`” is the function. Therefore, we found a direct match using our approach.

3.5 Result of Type Detection:

Our type detection process used the mapping module to bring the type of method calls with a return value from their definition block. We have chosen 5 Python projects that have 8,664 assignment operations, which were described in Figure 3.4. We found most of

Table 3.4: Expression Type of Value of Assignment

Expression Type of Value of Assignment	Count	Expression Type of Value of Assignment	Count
Call	3639	UnaryOp	72
Constant	1721	ListComp	50
BinOp	719	BoolOp	35
Name	664	JoinedStr	24
Subscript	585	IfExp	13
List	439	Compare	9
Attribute	297	DictComp	9
Dict	239	Lambda	4
Tuple	142	GeneratorExp	2
		SetComp	1

the method calls, which are 42% of the total value type in the subject system. However, we solved the basic types described in the section 3.2. Our approach successfully detects the type of all Constant, Tuple, Dict, List, JoinedStr, BoolOp, SetComp, DictComp, Subscript, and ListComp perfectly, leading to 3,204. Then, our mapping solved the issue of 3,639 method calls. Our mapping algorithm mapped 2,263(62%) method calls and generated the type of that assignment operation. On the other hand, we found 394 Name expressions out of 664 numbers of Names, which is 59% of the total Name expression. Besides, for attribute, we found 77% type information, and for BinOp, we found 56% accurate values. Therefore,

our model achieved a generation of types for 70% of the assignments.

3.6 Evaluation Procedure:

In this section, we will evaluate the effectiveness of our mapping algorithm in terms of accuracy (exact match of the method call to its definition) by manual study. Our evaluation of the mapping algorithm discovered the following research question-

- **RQ1:** How effective is our mapping algorithm for random examples from four different projects?

3.6.1 Experimental Setup

In our dataset, we have 2678 source files that have the extension “.py” or “.pyi”. We used Python AST (see for detail A.3) for extraction of method call and method definition. After extraction, we found 146k method calls, and by random selection, we selected 250 method calls. Before using the mapping algorithm, we used four individual environments and installed all the required libraries.

3.6.2 Evaluation Metrics

We used the manual analysis to investigate the accuracy of our mapping. On 146k method calls, we used our mapping algorithm and found that, on average, 3% of calls had multiple mapping. We considered them as a missed example. Therefore, we only considered the exact mapping of the method calls and their definitions.

$$Accuracy = \frac{Number\ of\ Exact\ Matched}{Total\ Number\ of\ Example} \quad (3.2)$$

3.7 Result of Mapping Algorithm

We took four Python projects for our study. We considered these as they are popular libraries and had “requirement.txt file”, with more than a 500-star count, more than two committers, and a working duration of more than one year - “pandas”³, “numpy”⁴, “scikit-image”⁵, “GitPython”⁶. We installed all the required libraries mentioned in the “requirement.txt file”. When we installed these libraries, we collected all the environment paths. As for mapping, we need the required paths to find the method definitions. While analyzing the path variables, different path values for multiple created environments were used.

³<https://github.com/pandas-dev/pandas>

⁴<https://github.com/numpy/numpy>

⁵<https://github.com/scikit-image/scikit-image>

⁶<https://github.com/gitpython-developers/GitPython>

Therefore, we configured the environment for each library and used our approach to check the performance of the mapping algorithm. We found the method calls are related to seven different path variables.

- **Built-in method definitions and class methods** found in `-\\python_stubs\\1995209621\\builtins.py` which contains built-in classes and functions such as - “len”, “min”, “open”, “print”. It comes with the basic installation of Python.
- **Exception and Error Handling** We found some of the method calls are class objects, which are “Error ”, “Exception ”, “write” and “read”.
- **Installed Library Path:** Rest of the method definitions are found in the installed library. When we install a library via “pip3” it is added to a certain directory known as `\\Lib\\site-packages`. We used our mapping algorithm to search for the method definition in this path based on the library’s name and its calls.

Table 3.5: Manual Investigation Report of Mapping of Method Calls and their Definitions (250 examples for each project)

Project Name	Accuracy(%)
pandas	83
numpy	81
scikit-image	85
GitPython	83

We collected 146k method calls and used our mapping algorithm. To find the efficiency of our algorithm, we picked the 250 method calls to form each project. We verified the performance of our model by manually checking the mapping of 1000 examples. First, we opened up each example and checked the method definitions brought with the method calls. We considered those method calls with a single mapping of method definitions. The accuracy is shown in Table 3.5. We found our algorithm had an accurate mapping of 81%-85% cases on average.

3.8 Conclusion

Our import statement analyzer and mapping process solved two major issues. They can be described in short-

- The process brings the method definition to make the binder for method calls.
- For type solving of a variable, 42% of the values of assignments are from method calls, and the types are solely dependent on the return value of a method definition.

Therefore, our approach provides, on average, 80% correct mapping of the method calls and their definitions.

In chapter 4, we replicated a study called DeepBugs, where we needed to make a tuple of method calls and their definitions. Our study showed that 70% of method calls are connected with the library or any method definitions in the same project but in different files. Therefore, it is mandatory to find those definitions. On the other hand, we found approximately 42% of the values of assignments were from method calls. Therefore, to replicate the DeepBugs and find the type of method calls of assignments, we used our approach, which increased the mapped number of examples on average 15% more than the DeepBugs.

Chapter 4

Exploring Name-based Bug Detection in Python

4.1 Introduction

The names of identifiers (i.e., variables and methods) provide information about the semantics of programs. Descriptive and meaningful identifier names not only assist in program comprehension but can provide additional sources of information to support software development tasks such as completing code, suggesting identifier names, and detecting incorrectly ordered arguments of the same type. For statically typed languages (like C, C++, and Java), the type information of variables is embedded in the source code itself, which can help to detect incorrectly ordered arguments of different types. However, to detect incorrectly ordered arguments of the same type, we need to rely on other information from the source code. The problem is exacerbated in dynamically typed languages where type information is not even available. The similarity between argument and method parameters and the usage patterns of arguments can be leveraged to detect bugs caused by incorrectly ordered arguments. A prior study by Pradel et al. [2] showed that we can leverage the identifier names to develop a learning-based approach called DeepBugs that can detect argument selection defects. The approach differs from other name-based bug detection techniques because it does not require manually crafted rules to set argument values. The approach was evaluated using the 150K JavaScript dataset. However, we found a gap in how they performed their analysis. Firstly, the technique was evaluated using source code written in JavaScript. It is not clear whether the performance of the technique can be generalized to other programming languages. Our selection of Python is based on the fact that it is dynamically typed and the most popular language in the TIOBE index. Secondly, the study only considers the method calls and their definitions in the same file. During our study on the Python dataset, we found that the definitions of more than 70% of all method calls that have more than one argument

```

1724 def prune_vocab(vocab, min_reduce, trim_rule=None):
1725     """Remove all entries from the `vocab` dictionary with count smaller than `min_reduce`. ...
1744     result = 0
1745     old_len = len(vocab)
1746     for w in list(vocab): # make a copy of dict's keys
1747         if not keep_vocab_item(w, vocab[w], min_reduce, trim_rule): # vocab[w] <= min_reduce:
1748             result += vocab[w]
1749             del vocab[w]
1750     logger.info(
1751         "pruned out %i tokens with count <=%i (before %i, after %i)",
1752         old_len - len(vocab), min_reduce, old_len, len(vocab)
1753     )
1754     return result

422 class Phrases(_PhrasesTransformation):
423     """Detect phrases based on collocation counts."""
424
425     def __init__(...
571
572     def __str__(self):...
577     @staticmethod
578
579     def learn_vocab(sentences, max_vocab_size, delimiter, connector_words, progress_per):
580         sentence_no, total_words, min_reduce = -1, 0, 1
581         vocab = {}
582         logger.info("collecting all words and their counts")
583         for sentence_no, sentence in enumerate(sentences):
584             if sentence_no % progress_per == 0:
585                 logger.info(
586                     "PROGRESS: at sentence #%i, processed %i words and %i word types",
587                     sentence_no, total_words, len(vocab),
588                 )
589             start_token, in_between = None, []
590             for word in sentence:
591                 if word not in connector_words:
592                     vocab[word] = vocab.get(word, 0) + 1
593                     if start_token is not None:
594                         phrase_tokens = itertools.chain([start_token], in_between, [word])
595                         joined_phrase_token = delimiter.join(phrase_tokens)
596                         vocab[joined_phrase_token] = vocab.get(joined_phrase_token, 0) + 1
597                     start_token, in_between = word, [] # treat word as both end of a phrase AND beginning of another
598                 elif start_token is not None:
599                     in_between.append(word)
600             total_words += 1
601         if len(vocab) > max_vocab_size:
602             utils.prune_vocab(vocab, min_reduce)
603             min_reduce += 1

```

Figure 4.1: Method Call and its Mapped Definition

are located in different files. It was not clear how that could affect the performance of the technique. Thirdly and most importantly, there are a large number of library method calls whose definitions are not present in the source code. For example, for our Python dataset, 86% of method calls are related to libraries, and 14% of those calls have more than one argument. DeepBugs did not consider those method calls in their study as the definitions of those calls are not located in the same repository where the method calls are located. This motivated us to investigate the problem further, focus on identifying the performance of DeepBugs for other programming languages, and investigate the importance of method definitions for detecting bugs caused by incorrectly ordered arguments. A method call may have multiple number of arguments. The value of the arguments comes from the variables assigned in the source code before the line it used by calling a method or using an object of a class. The variations of expression of an argument are described in Table 4.1. However, the parameter and the method definition body decide which type of argument can be used for the method call. We proposed a model to collect all the usage context of the argument and used them to train a deep learning model. Our model did not use the method definition as the method definition may not be available in the environment. Therefore, our

model detects an argument swap by considering the name of a method call and the usage context of an argument. Besides, from our model, the programmer will understand how to write down a method body to consider the correct sequence of arguments. This usage pattern of the method arguments is categorized into four types described in the section 4.2. This chapter proposed a name-based bug detection tool using the argument usage context from a given source code. We replicated the study of an existing approach, DeepBugs, for Python and compared DeepBugs with our proposed model. We used 150k Python files for our study and a Python AST parser for data extraction. The extracted data was used for mapping method calls with their method definitions, context collection, generating correct and wrong code sequences, and type detection. After collecting the information, we trained models and evaluated our model by comparing the performance (accuracy, recall, precision) with DeepBugs on Python. Next, we checked the performance of mapped and unmapped method calls for both models. This showed the importance of mapping argument-related bug detection. After comparing our model with DeepBugs, we concluded that our model performed better than DeepBugs Model. This required an empirical study to find the reason for getting high accuracy. This led us to conduct a qualitative study to discover the reason for better performance. We combined different context-based models and showed how the context information led to a better result. The trained model with only mapped method calls and the model trained with all method calls performed almost the same, and the proposed approach can be used for any large-scale study. Moreover, we checked the overall and expression type-wise performance to ascertain the reason behind the variation of the result. Our study also compared the importance of context for the detection of argument swapping by comparing our model with an existing pre-trained model (CodeBert). Our model outperformed the model (trained from the vectors generated by CodeBert). We evaluated our model with the data extracted from 150k Python files. Our model has an accuracy of 90.79% with an 88% precision score. The contributions of this chapter are as follows:

- Replicating the existing bug detector model for Python and Comparing it with our proposed model. Both models considered the semantic information of tokens from source code to determine the argument-related bugs for dynamically typed language.
- Showing the importance of different source information for training the model.
- Comparing with a model trained from a pre-trained model and our deep learning model.
- Introducing variable and argument usage context for bug detection.

Thus, we structured the chapter as follows.

- section 4.2 described the terminologies we have used and the background of our study.
- section 4.3 contains the description of our dataset.
- We described the study procedure, experimental Setup, evaluation procedure and results of our research questions in Sections 4.4, 4.5 and 4.6.
- We discussed threats to the validity of this study in section 4.7.
- Finally, Section 4.8 concludes the thesis and future research directions.

4.2 Background

This section defines the useful terms and context definition to understand the methodology easily.

- **Swapping Argument Issues in Python:** A method call can take multiple arguments for their method definitions. In this method, arguments are mostly defined or assigned before they are used as arguments. This argument should pass in the correct order or sequence to properly use its corresponding method definition. Therefore, annotating the method argument is mandatory for all programming languages. To describe our intention of the research, we took a well-known library project, “gensim,” where the “phrases.py” file has a method call at line 602 “utils.prune_vocab” with its two arguments “vocab” and “min_reduce”. In Figure 4.1, the method call is defined in the “utils.py” file at line 1724. After getting the mapping with the method call and its definition, this dynamically typed language will not show any error or warning even if we swap the order of the arguments from `utils.prune_vocab(vocab, min_reduce)` to `utils.prune_vocab(min_reduce, vocab)` at the compile time. When we run the program, it will generate an error showing an error or warning that “data type inconsistent or value error”. Therefore, the user and the programmer are unaware that they have already swapped the argument, which may cause an error. We aim to collect the static information and build a model to identify this error during code writing.
- **Method Global Mapping:** A method call must be mapped with its corresponding method definitions. Though this mapping method is challenging for all compilers, we proposed an alternative method to overcome this issue. In Figure 4.1, the method call “`utils.prune_vocab(vocab, min_reduce)`” has its definition in a different file. In the DeepBugs model, they mapped those method calls with a method definition that is only in the same file. However, Python has a convention of calling methods from other files through import statements. These import statements can import method definitions from the same projects or the predefined library definitions. Following the

DeepBugs approach, we missed maximum method calls and definitions. Therefore, we analyzed the import statements of each file and brought inter-project method declarations in that file scope. The example in Figure 1 will be missed as the method declarations are in “utils.py” and the method calls in “phrases.py” are in two different files.

- Local Context(LC):** We collected the method call and its local contexts. A method call and its surrounding ten tokens after and before are considered as local context. Therefore, a local context consists of a total number of twenty-one tokens. When we generate vectors for the name of a method call and its argument, we consider this local context as we believe similar source code elements will appear in a similar context [2]. Therefore, the method call and its argument will follow a similar context. From Figure 4.1, the local context for the argument “vocab ” from “utils.prune_vocab(vocab, min_reduce)” will be [‘min_reduce’, ‘min_reduce’, ‘logger’, ‘info’, ‘len’, ‘vocab’, ‘total_words’, ‘sentence_no’, ‘utils’, ‘prune_vocab’, ‘min_reduce’, ‘vocab’, ‘return’, ‘min_reduce’, ‘vocab’, ‘total_words’, ‘def’, ‘add_vocab’ ‘self’ ‘sentences’] (after removal of unnecessary tokens).
- Argument Usage Context(AUC):** To use an argument in a method call, the programmer needs to declare it before the line it will use. The recent usage of the method argument as a variable or any assignment is called the usage context. In Figure 4.1, the method call “utils.prune_vocab(vocab, min_reduce)” has an argument for “vocab” which has usage at lines 601, 596, 592, and 581. In line 581, it has an expression as a dictionary definition; at lines 592 and 596, it was used as an access to the dictionary; at line 601, “vocab” is used as an argument of a method call. We have collected all the usage data, though we ignored the usage outside the method definition. As the variable “vocab” has scope inside the method definition “_learn_vocab” at line number 579, we did not collect the context for “vocab” outside the method definition. Therefore, for the word “vocab,” the collected context will be the usage in the field scope and the line of the method definition. Thus the context will be [[‘def’ ‘_learn_vocab’ ‘sentences’ ‘max_vocab_size’ ‘delimiter’ ‘connector_words’ ‘progress_per’],[‘vocab’],[‘vocab’ ‘word’ ‘vocab’ ‘get’, ‘word’],[‘vocab’ ‘joined_phrase_token’ ‘vocab’ ‘get’ ‘joined_phrase_token’],[‘if’ ‘len’ ‘vocab’ ‘max_vocab_size’]].
- Line Context:** We used the usage context of arguments as features for our model. The line context implies all the tokens of a particular line. As an example of line context at line no. 581 in Figure 4.1 is [“vocab”, “word”, “vocab”, “get”, “word”, “0”, “1”].

Table 4.1: Generalization of Extracted Argument Names

Real Time Example of Arguments	Extracted Name	Expression Type
<code>make_pymodule_path(filename)</code>	<code>make_pymodule_path</code>	Call
<code>args . command</code>	<code>command</code>	Attribute
<code>_name_</code>	<code>_name_</code>	Name
<code>[variables [i] [k] for k in li [j]]</code>	<code>constant</code>	ListComp
<code>[“ -username=bob” , “ -password=bar”]</code>	<code>constant</code>	List
<code>entry [“author”]</code>	<code>entry</code>	Subscript
<code>lambda x , y : map(lambda z : z / y , x) , inv , l)</code>	<code>Lambda</code>	Lambda
<code>* matchers</code>	<code>matchers</code>	Starred
<code>(f” {versionfile source}export-subst\n”)</code>	<code>constant</code>	JoinedStr
<code>(“Admin” , “admin@localhost.local”)</code>	<code>constant</code>	Tuple
<code>- max abs</code>	<code>max abs</code>	UnaryOp
<code>t in selected types for t in(sort to opts (sorter . type) , 0)</code>	<code>GeneratorExp</code>	GeneratorExp
<code>{ key : (val if isinstance (val , list) else val . split ()) for key , val in queues . items () }</code>	<code>dictComp</code>	DictComp
<code>{ “secondary uom qty” : 2 , “secondary uom id” : self . secondary unit box 5 . id }</code>	<code>dict</code>	Dictionary
<code>“Attempted to decode corrupted number”</code>	<code>constant</code>	constant
<code>_pc name in set (settings . get (‘installed packages’ , [])</code>	<code>_pc name</code>	Compare
<code>2 , e2 := int (math . log2 (base)</code>	<code>NamedExpr</code>	NamedExpr
<code>(message or ”(no message)”)</code>	<code>”message or (no message)”</code>	BoolOp
<code>self . assertEqual (self . storage . filepath (x = 0 , y = 1 , z = 2 , hashed = 0xdeadbeef)</code>	<code>BinOp</code>	BinOp
<code>await get (context . query tree ())</code>	<code>await</code>	Await
<code>{ str }</code>	<code>constant</code>	Set
<code>{ field name . split (“ ” , 1) [0] for field name in self . all field names }</code>	<code>constant</code>	SetComp
<code>yield from resp . read ()</code>	<code>YieldFrom</code>	YieldFrom

- Parent Context:(PC)** A method call can be in the scope of a method definition, class definition, conditional statements, or any loop statement. As a parent block context, we collected the line context of a parent block and added it to the argument’s recent usage context. In Figure 4.1, the latest usage of the argument “min_reduce” is found at line no.580, and it is under the parent block at line no 579. therefore the parent context will be [“def” , “_learn_vocab” , “sentences” , “max_vocab_size” , “delimiter” , “connector_words” , “progress_per”] and [“sentence_no” , “total_words” , “min_reduce” , “1” , “0” , “1”]. These contexts were used to train the vector model to build the actual usage pattern

of the method argument.

- Context Pre-processing:** The source code comprised all types of tokens. Though the tokens are important for executing the source code, for embedding generation, these tokens are not important. These tokens generate data redundancy and ambiguity in the code sequence. Therefore, we used a context pre-processor to remove some unnecessary tokens from the source code. In Figure 4.1, we considered the segment from line no 601 to 603 and collected the context as- ["utf8", "if", "len", "(", "vocab", ")", ">", "max_vocab_size", ":", "utils.prune_vocab(vocab, min_reduce)", " ", "utils", ".", "prune_vocab", "(", "min_reduce", ",", " ", "vocab", ")", "min_reduce", "+=", "1"]. These tokens are refined to ["if", "len", "vocab", "max_vocab_size", "utils", "prune_vocab", "vocab", "min_reduce", "min_reduce", "1"].

4.3 Data Collection

For this study, we consider a collection of Python files collected from open-source GitHub repositories. Our selection of the dataset is based on the fact that it is publicly available¹, consists of 150K Python files from 5958 repositories covering different domains, and is used by prior studies [44]. Our dataset may contain duplicate source files. A prior study

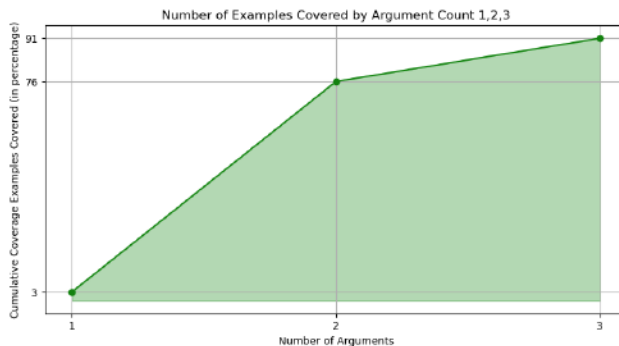


Figure 4.2: Number of Examples Covered by Argument Count 1,2,3

by Allamins et al. [45] showed that code duplication negatively impacts the performance of machine learning models on source code. We remove duplicate files from our dataset to compare machine learning models more accurately. We used a code-deduplication tool called CD4Py to detect near and exact duplicate Python source files². Duplicate files are detected as clusters, and each cluster represents duplicated files. CD4Py keeps one file from each cluster and removes the others. We removed 24,504 (19.58%) source files by applying

¹<http://files.srl.inf.ethz.ch/data/py150.tar.gz>

²<https://github.com/saltudelft/CD4Py>

the de-duplication tool. Our de-duplicated dataset consists of 125,496 source files. We used the Python standard AST parser(see section A.3) to parse and collect our required features from the source code.

4.4 Methodologies

4.4.1 Data Extraction and Generalization

To analyze and extract features from Python source code, we used Python AST parser³ and followed the process described at section A.3. We considered each Python file as a single source of information and passed it directly to the Python AST to generate an AST representation of the source code. This AST parser generated a node, a collection of objects of expression types. We extracted two features (noted as objects in the node) from each AST: i) Method Definition and ii) Method Call.

For the method definition, we collected the **method name**, **the list of parameters**, **the line number**, **the class information**, and **the file name** (where the method definition is declared). We also saved the whole node for further information for the function node body. For method calls, we collected the **method call name**, **the line number**, **the file name** where it was called, the list of arguments, and the **method call's parent block information**. In addition, we collected the local context by ten tokens after and before the position [2] of the argument in the method call.

Then, we collected the usage pattern of the arguments by extracting the latest appearance of those arguments in the same scope in the current file. An argument must be defined before it is used; therefore, it has a validation of usage in the same scope it was declared. For this step, we used the parent block information to collect the context by the local and global usage patterns. To collect this usage pattern and appearance, we tokenized the whole file and analyzed them with a name-based approach [26] [36]. To collect all the above information, we first tokenized the source code files and then used a name-based approach to locate the code's definitions, method calls, and tokens of their feature.

When we dug deeper into the script, we found that the arguments of a method call are mostly a variable, a method call, or any expression type (see Table 4.1 to understand different expression types). To track down the variable's value where assigned, we extracted all the assignments related to that variable inside the project files. Besides, to map the method calls with their respective method declarations in the source code, we parsed the import statements of each Python script. These import statements implied how source codes were connected and how their contents could be reused in another source code. We discussed earlier in section 3.3.4 about the pattern of import statements.

³<https://docs.python.org/3/library/ast.html>

We collected all the expression types for the method arguments. At DeepBugs- [2], they collected limited expression types, whereas we handled all the expression types described at the Python AST⁴. We collected and generalized the tokens while working on name-based analysis. Table 4.1 shows how we have collected the information from each argument. Therefore, the generalized segment was treated as the actual name of the identifier, and these were considered as the names of the features. We used those arguments that had arguments more than 1. In Figure 4.2, we plotted the coverage of examples by their count of arguments per example and found almost 91% of examples were covered by examples that had many arguments less than 4. **Therefore we conducted our study on swapping arguments of first two adjunct arguments.**

4.4.2 Linking Method Calls to its Definitions

Linking method calls to their definitions are required to support the implementation of DeepBugs, a state-of-the-art name-based bug detection technique. DeepBugs utilizes semantic vector representation of arguments of a method call and the formal parameters of the called method. We implement DeepBugs for the Python programming language to compare with our proposed technique. We followed the steps from the section 3.3. Though our section 3.3 considered all of the method definitions from all the libraries and built-in functions, we considered mapping of method calls from the same project for our study. To compare with the existing approach (DeepBugs), we needed only those method calls that had a mapping in the same file. Therefore, we collected the mapped method calls only for the same project. After using the approach described in section 3.3, we found that 1,76,743 method calls were mapped successfully within the same project. These examples were used to train the DeepBugs model to evaluate the performance of Python Language.

4.4.3 Binder Generation

After the method call and its definition collection, we changed the words to their vectors by using the gensim word2vec⁵ model. The Word2vec model will go through the context and generate vectors from the code context. **The binder is the collection of call and definition features that will be used to detect swapping argument-related bugs.**

4.4.3.1 Binder Generation for Correct Code Pattern

We have four different binders for four different models. This binder consists of the features used in the model. Additionally, this binder indicates how the model should learn the correct and wrong sequence. Our first model is the original DeepBugs model with the

⁴<https://docs.python.org/3/library/ast.html>

⁵<https://radimrehurek.com/gensim/models/word2vec.html>

Table 4.2: Binder Information for DeepBugs Model(DBM) and API UsageBased Model (AUM)

Notation	Description	Context Notation	Collected Context
MC_N	Method Call Name	MC_N_LC	["in between", "append", "word", "total words", "1", "if", "len", "vocab", "max vocab_size", "utils", "prune_vocab", "vocab", "min_reduce", "min_reduce", "1", "logger", "info", "len", "vocab", "total words", "sentence no"]
MD_N	Method Definition Name	MD_N	["def", "prune_vocab", "vocab", "min_reduce", "trim_rule", "None", "result", "0", "old_len", "len", "vocab"]
MD_{P_1}	Parameter 1 of Method Definition	MD_{P_1}	
MD_{P_2}	Parameter 2 of Method Definition	MD_{P_2}	
MC_{A_1}	Argument 1 of Method Call	$MC_{A_1_LC}$	["append", "word", "total words", "1", "if", "len", "vocab", "max vocab_size", "utils", "prune_vocab", "vocab", "min reduce", "min reduce", "1", "logger", "info", "len", "vocab", "total words", "sentence no", "1"]
MC_{A_2}	Argument 2 of Method Call	$MC_{A_2_LC}$	["word", "total words", "1", "if", "len", "vocab", "max vocab_size", "utils", "prune_vocab", "vocab", "min reduce", "min reduce", "1", "logger", "info", "len", "vocab", "total words", "sentence no", "1", "return"]
$MC_{A_1_Type}$	Type of Argument 1 of Method Call	$MC_{A_1_Type}$	Name
$MC_{A_2_Type}$	Type of Argument 2 of Method Call	$MC_{A_2_Type}$	Name

following features in the binder:

Binder Generation For DeepBugs Model for Python(DBM): A correct code pattern was gained by parsing the source code files and collecting the features for method definitions and calls. After mapping the method call with the method definition, a correct code pattern was generated by combining the features denoted at Table 4.1 in a certain sequence. (see Table 4.2 for Abbreviation of the notations). Therefore, the correct code pattern for the DeepBugs Model is addressed in the equation 4.1.

$$CorrectCodePattern(DBM) = (MD_N, MC_N, MC_{A_1}, MC_{A_2}, MC_{A_1_Type}, MC_{A_2_Type}, MD_{P_1}, MD_{P_2}) \quad (4.1)$$

In the example in Figure 4.1 the binder formation is [“prune_vocab”, “prune_vocab”, “vocab”, “min_reduce”, “Name”, “Name”, “vocab”, “min_reduce”] for the method call “utils.prune_vocab(vocab,min_reduce)”

Binder Generation for Argument Usage Pattern with Parent Information and Expression Type Information: Our study is based on the usage context of the argument. We proposed a solution to detect the argument-swapping-related bugs even if the method call information is missing. Therefore, we generated three more binders for three models where we did not consider the method definition. They are described below-

- **Binder Generation for API Usage-Context :** We built a binder for the API Usage-Context Model(AUM) for the first usage-based model. In this model, we considered only API features. Our goal was to check the performance of the DeepBugs Model to see if there was missing information on the method Definition. Therefore, for this model, the correct code binder with notation(see Table 4.2 for Abbreviation of the notations) was shown by the equation 4.2.

$$CorrectCodePattern(AUM) = (MC_N, MC_{A_1}, MC_{A_2}, MC_{A_1_Type}, MC_{A_2_Type}) \quad (4.2)$$

Table 4.3: Binder Information for AUCM Model

Notation	Description
MC _N	Method Call Name
MC _{A₁} _LC	Local Context of Argument 1 of Method Call
MC _{A₂} _LC	Local Context of Argument 2 of Method Call
MC _{A₁} _Type	Type of Argument 1 of Method Call
MC _{A₂} _Type	Type of Argument 2 of Method Call
MC _{A₁} _PCLU	Parent Block and latest usage of Arg 1 of Method Call
MC _{A₂} _PCLU	Parent Block and latest usage of Arg 2 of Method Call
MC _{A₁} _AUC	All usage Context of Arg 1 of Method Call
MC _{A₂} _AUC	All usage Context of Arg 2 of Method Call

In the example in Figure 4.1, for the method call “utils.prune_vocab(vocab,min_reduce)”, the binder formation is [“prune_vocab”, “vocab”, “min_reduce”, “Name”, “Name”].

This model only considered the method call name and its argument. When collecting the context for this binder, we collected the local context of method call and its argument.

- **Binder Generation for Argument Usage Pattern with Parent Information:**

Table 4.4: Collected Context for AUCM Model

Binder Features	Token Name	Line No.	Context Remark	Collected Context
MC _N _LC	prune_vocab	602	Local Context	["in_between", "append", "word", "total_words", "1", "if", "len", "vocab", "max_vocab_size", "utils", "prune_vocab", "vocab", "min_reduce", "min_reduce", "1", "logger", "info", "len", "vocab", "total_words", "sentence_no"]
MC _{A1} _LC	vocab	602	Local Context	["append", "word", "total_words", "1", "if", "len", "vocab", "max_vocab_size", "utils", "prune_vocab", "vocab", "min_reduce", "min_reduce", "1", "logger", "info", "len", "vocab", "total_words", "sentence_no", "1"]
MC _{A1} _LC	min_reduce	602	Local Context	["word", "total_words", "1", "if", "len", "vocab", "max_vocab_size", "utils", "prune_vocab", "vocab", "min_reduce", "min_reduce", "1", "logger", "info", "len", "vocab", "total_words", "sentence_no", "1", "return"]
MC _{A1} _Type	vocab	602		Name
MC _{A2} _Type	min_reduce	602		Name
MC _{A1} _PC_LU	vocab	583	Parent Line Context	["for", "sentence_no", "sentence", "in", "enumerate", "sentences"]
		601	Line Context	["if", "len", "vocab", "max_vocab_size"]
MC _{A2} _PC_LU	min_reduce	579	Parent Line Context	["def", "learn_vocab", "sentences", "max_vocab_size", "delimiter", "connector_words", "progress_per"]
		580	Line Context	["sentence_no", "total_words", "min_reduce", "1", "0", "1"]
MC _{A1} _AUC	vocab	581	Line Context	["vocab", "word", "vocab", "get", "word", "0", "1"]
		592	Line Context	["vocab", "word", "vocab", "get", "word", "0", "1"]
		596	Line Context	["vocab", "joined_phrase_token", "vocab", "get", "joined_phrase_token", "0", "1"]
		601	Line Context	["if", "len", "vocab", "max_vocab_size"]
MC _{A2} _AUC	min_reduce	580	Line Context	["sentence_no", "total_words", "min_reduce", "1", "0", "1"]

We extracted context based on the argument usage (Argument Usage Context-Based Model(AUCM)) to train the model with a better context. The information of an argument of a method call is embedded in the context of the usage of that argument. When we collected the latest usage of the argument, we collected the parent block information and the latest usage of the name. The context of the parent block will determine the argument’s usage based on their tokenized information. Besides, we have collected the local context by extracting the information from 10 tokens after and ten tokens before the argument in the method call. Next, we added the scope of the variable information to the usage context to uniquely identify the local and global variables. Therefore, the correct code binder for the argument usage context of the method call will be in the section 4.3 (see Table 4.3 for Abbreviation of the notations)-

$$\begin{aligned} \text{CorrectCodePattern}(AUCM) = & (MC_N, MC_{A_1_LC}, MC_{A_2_LC}, MC_{A_1_Type}, \\ & MC_{A_2_Type}, MC_{A_1_PC_LU}, MC_{A_2_PC_LU}, MC_{A_1_AUC}, MC_{A_2_AUC}) \end{aligned} \quad (4.3)$$

- **Binder Generation for Argument Usage Pattern with Parent Information and Expression Type Information(AUCMET):** Based on the Argument Usage Pattern with Parent Information, our second model suffered from the ambiguity of tokens and uncontrolled vocabulary length. Therefore, we introduced the expression type-based additional context for the **Argument Usage Context Model with Expression Type(AUCMET)**. This expression type was considered structural information. By adding the expression type information, we added the actual grammar of a programming language. The model trained from this feature got enough information about the correct sequence of every token. At the time of context collection, we programmatically and implicitly added the expression type of each token to identify the tokens uniquely of identical names. This expression-type information was collected by parsing every source code node and individually adding by considering a token’s column offset. Therefore, our model got additional information for a better understanding of the usage of tokens and correct usage patterns from the code segment to capture the argument swapping.

For our model based on Argument Usage Pattern with Parent Information and Expression Type Information, our binder formation is described in 4.4(see Table 4.5 for Abbreviation of the notations)

$$\begin{aligned} \text{CorrectCodePattern}(AUCMET) = & (MC_N_LC_ET, MC_{A_1_LC_ET}, \\ & MC_{A_2_LC_ET}, MC_{A_1_Type}, MC_{A_2_Type}, MC_{A_1_PC_LU_ET}, \\ & MC_{A_2_PC_LU_ET}, MC_{A_1_AUC_ET}, MC_{A_2_AUC_ET}) \end{aligned} \quad (4.4)$$

Table 4.5: Collected Context for AUCMET

Binder Features	Token Name	Line No.	Context Remark	Collected Context
MC _N _LC_ET	prune_vocab	602	Local Context	["in_between#call", "append#call", "word#arg", "total_words#assign", "1#constant", "if#IfExpr", "len#call", "vocab#arg", "max_vocab_size#unknown", "utils#call", "prune_vocab#call", "vocab#arg", "min_reduce#arg", "min_reduce#assign", "1#constant", "logger#call", "info#call", "len#call", "vocab#arg", "total_words#arg", "sentence_no#arg"]
MC _{A1} _LC_ET	vocab	602	Local Context	["append#call", "word#arg", "total_words#assign", "1#constant", "if#IfExpr", "len#call", "vocab#arg", "max_vocab_size#unknown", "utils#call", "prune_vocab#call", "vocab#arg", "min_reduce#arg", "min_reduce#assign", "1#constant", "logger#call", "info#call", "len#call", "vocab#arg", "total_words#arg", "sentence_no#arg", "1#constant"]
MC _{A2} _LC_ET	min_reduce	602	Local Context	['word#arg', "total_words#assign", "1#constant", "if#IfExpr", "len#call", "vocab#arg", "max_vocab_size#unknown", "utils#call", "prune_vocab#call", "vocab#arg", "min_reduce#arg", "min_reduce#assign", "1#constant", "logger#call", "info#call", "len#call", "vocab#arg", "total_words#arg", "sentence_no#arg", "1#constant", "return#return"]
MC _{A1} _Type	vocab	602		Name
MC _{A2} _Type	min_reduce	602		Name

Table 4.6: Collected Context for Latest and Parent Block Usage Context with Expression type information (Contd.)

Binder Features	Token Name	Line No	Context Remark	Collected Context
MC _{A1} _PCLU_ET	vocab	583	Parent Line Context	["for#For", "sentence_no#unknown", "sentence#unknown," keyword", 'enumerate#call', "sentences#arg']
		601	Line Context	["if#IfExp", "len#call", "vocab#arg", "max vocab_size#Name"]
MC _{A2} _PCLU_ET	min_reduce	579	Parent Line Context	[def#FuncDef, " learn_vocab#FuncDef", "sentences#par", "max_vocab size#par", "delimiter#par", "connector_words#par", "progress_per#par"]
		580	Line Context	["sentence no#assign", "total words #assign", "min reduce#assign", "1#constant", "0#constant", "1#constant"]
MC _{A1} _AUC_ET	vocab	581	Line Context	"vocab" #assign
		592	Line Context	["vocab#subscript", "word#unknown", "vocab#call", "get#call", "word#arg", "0#arg", "1#arg"]
		596	Line Context	["vocab#subscript", "joined phrase_token#unknown", "vocab#call", "get#call", "joined phrase_token #arg", "0#arg", "1##constant"]
		601	Line Context	["if#IfExpr", "len#call", "vocab #arg", "max vocab size#unknown"]
MC _{A2} _AUC_ET	min_reduce	5802	Line Context	["sentence no#assign", "total words##assign", "min reduce#assign", "1#constant", "0#constant", "1#constant"]

4.4.3.2 Swapping Argument Sequence for Wrong Code Pattern

A Buggy code pattern can not be directly collected from the repository. Therefore, we generated the wrong code pattern programmatically. According to DeepBugs [2], programmers are prone to errors for mixing up the arguments' sequence while calling the module. Besides, correct argument passing is mainly based on the number of arguments and the values passed at the time of the call. We have swapped the sequence of the actual argument and generated the buggy code sequence, and we followed the same context collected and described at Table 4.2. The generated wrong binder for DeepBugs Model is described in equation 4.5.

$$\begin{aligned} BuggyCodePattern(DBMW) = (MD_N, MC_N, MC_{A_2}, MC_{A_1}, \\ MC_{A_2_Type}, MC_{A_1_Type}, MD_{P_1}, MD_{P_2}) \end{aligned} \quad (4.5)$$

Therefore the buggy code binder for the method call “utils.prune_vocab(vocab,min_reduce)” in Figure 4.1 will be [“prune_vocab”, “prune_vocab”, “min_reduce”, “vocab”, “Name”, “Name”, “vocab”, “min_reduce”].

Similarly, we generated a buggy code sequence for the API Usage-Context model(AUM) by swapping two arguments, as described in the equation 4.6.

$$BuggyCodePattern(AUMW) = (MC_N, MC_{A_2}, MC_{A_1}, MC_{A_2_Type}, MC_{A_1_Type}) \quad (4.6)$$

For the AUCM Model, we followed a similar approach to generate the buggy sequence, and the formation (equation 4.7) of the AUCM is as follows:

$$\begin{aligned} BuggyCodePattern(AUCMW) = (MC_N, MC_{A_2_LC}, MC_{A_1_LC}, MC_{A_2_Type}, \\ MC_{A_1_Type}, MC_{A_2_PC_LU}, MC_{A_1_PC_LU}, MC_{A_2_AUC}, MC_{A_1_AUC}) \end{aligned} \quad (4.7)$$

For our proposed model, we have used the binder formation at section 4.8, which considered the extended information by augmenting Expression type information with tokens.

$$\begin{aligned} BuggyCodePattern(AUCMETW) = (MC_N, MC_{A_2_LC_ET}, MC_{A_1_LC_ET}, MC_{A_2_Type}, \\ MC_{A_1_Type}, MC_{A_2_PC_LU_ET}, MC_{A_1_PC_LU_ET}, MC_{A_2_AUC_ET}, MC_{A_1_AUC_ET}) \end{aligned} \quad (4.8)$$

4.4.4 Context Collection for Word2vec Model

We collected context and tokenized them using a Python tokenizer [46]. We considered ten tokens after and before the particular word and all the tokens from a line when collecting

the tokens. We described four types of context information in section 4.2. Now, according to the requirements of binders for different models (described in section 4.4.3). Therefore, our processed dataset was generated for four different models.

Table 4.7: Encoding of Tokens for AUCMET

Example	Expression Type	Encoding
def _learn_vocab	FunctionDef	_learn_vocab#FunctionDef
def _learn_vocab(sentences)	para	sentences#para
vocab = {}	Assign	vocab#Assign
count=1	Constant	1#Constant
vocab[word]	Subscript	vocab#Subscript
info	call	info#call
if word not in connector_words	IfExpr	if#IfExpr
unigram + bigrams	binop	unigram#binop
if word not in connector_words	compare	word#compare
c={1,2}	Dict	{1,2}#Dict
class student:	ClassDef	class#ClassDef
class student	ClassDef	student#ClassDef
unittest.test()	Attribute	unittest#Attribute
unittest.test()	call	test#call
def learn_vocab(*sentences)	stared	sentences#stared
i=true	boolean	true#boolean
any other tokens	unknown	tok_name#unknown

4.4.4.1 Context Collection for DeepBugs Model(DBM)

DeepBugs Model required eight features, and Table 4.2 showed the collected context for the model. We followed the equation 4.1 and 4.5, and the collected context was denoted here in Table 4.2. We collected only those tokens after the “def” token for method definition. Programmers can declare a function at any location of a source code. Therefore, it is not mandatory to collect local context. Therefore, we only collected the information from the method definition body. These contexts were trained individually to a word2vec model. Therefore, we trained a new word2vec model for every new example and collected the vector from the model. We used the word2vec model eight times for each example to train and collect vectors for each feature. We wanted to compare DeepBugs and the AUCMET with all method calls and only the mapped method calls; we used a constant vector of identical

length and filled in those examples where method definitions were missing. This helped us consider all the examples and check the performance of DeepBugs.

4.4.4.2 Context Collection for API Usage-Context Mode(AUM)

The firm difference between the DBM model and the AUM Model is the removal of the method definition. Therefore, we used the same context of the DBM Model after removing features from the definition of the method. We followed the equation 4.2 and 4.6 for collecting the contexts as per the description in section 4.2. The collected contexts were at Table 4.2. Thus, we generated the vector for the DeepBugs Model and removed the definition vector to generate data for the AUM model.

4.4.4.3 Context Collection for Argument Usage Pattern with Parent Information Model(AUCM Model)

For all Method calls, we collected additional information, and these features were described in Table 4.3. The collected context was described in Table 4.4. An argument must have to be declared and assigned before it is used. Therefore, this model was built on top of how an argument was used before a line of method call. As we did not provide any information about the method definition, this model is suitable even if the method definition is missing.

For the Argument Usage Pattern with the Parent Information Model, we combine four different features for each argument. First, we collected the line context of each argument. Then, we checked which parent block the method call is situated. From the code segment in Figure 4.1, for the first argument “vocab” of the method call “prune_vocab,” the argument usage context found at lines no.581, 592, 596, 601 and the “min_reduce” (at lines no.582) before line 602. Our analysis determined the correct position of argument based on their usage pattern. For the argument’s latest usage context with its parent block information, for the first argument, “vocab” of the method call “prune_vocab”, we collected context from line number 583 as parent block information and context from line no 601 as argument latest usage context. We collected the type of each argument by using the approach described in section 3.2; therefore, for the first argument, “vocab,” the expression type is “Name.”

4.4.4.4 Context Collection for Argument Usage Pattern with Parent Information and Expression Type Information(AUCMET)

Similarly, we augmented the expression type expression with Context of Argument Usage Pattern with Parent Information to collect the additional information for our model. Table 4.5 showed the collected context for the Argument Usage Pattern with Parent Information and Expression Type Information Model. Our tokenization process generated the token,

```

422 class Phrases(_PhrasesTransformation):
423     """Detect phrases based on collocation counts."""
424
425     def __init__(...
571
572     def __str__(self):...
577     @staticmethod
578
579     def _learn_vocab(sentences, max_vocab_size, delimiter, connector_words, progress_per):
580         sentence_no, total_words, min_reduce = -1, 0, 1
581         vocab = {}
582         logger.info("collecting all words and their counts")
583         for sentence_no, sentence in enumerate(sentences):
584             if sentence_no % progress_per == 0:
585                 logger.info(
586                     "PROGRESS: at sentence #%i, processed %i words and %i word types",
587                     sentence_no, total_words, len(vocab)
588                 )
589                 start_token, in_between = None, []
590                 for word in sentence:
591                     if word not in connector_words:
592                         vocab[word] = vocab.get(word, 0) + 1
593                         if start_token is not None:
594                             phrase_tokens = itertools.chain([start_token], in_between, [word])
595                             joined_phrase_token = delimiter.join(phrase_tokens)
596                             vocab[joined_phrase_token] = vocab.get(joined_phrase_token, 0) + 1
597                             start_token, in_between = word, [] # treat word as both end of a phrase AND beginning of another
598                         elif start_token is not None:
599                             in_between.append(word)
600                 total_words += 1
601                 if len(vocab) > max_vocab_size:
602                     utils.prune_vocab(vocab, min_reduce)
603                     min_reduce += 1

```

Figure 4.3: Encoding of A Word Based on Different Expression Type

and the AST parser provided the expression type information for the token. Table 4.7 included the embedding process of expression type. We considered only those tokens related to the method call, their definitions, and their features. As arguments were related to the assignment expression, we considered the assignment expression, too. The data flow of a source code was related to control and conditional statements, such as “if”, “for”, “elif” which were also considered in the embedding. On the other hand, as an argument is most likely to be a name, constant, attribute examples (see Table 5.2 for argument expression type frequency), we considered all the embedding of a name and its usage perfectly. Therefore, the same name can be embedded into an argument, name, call, or constant by the place used.

Example of Encoding for a Token “vocab”: The token “vocab” was found in line no. 581, 587, 592, 596, 601, and 603 in the Figure 4.3. Therefore, our encoding generator built four encoding types for the token “vocab”. This helped our model to learn a specific pattern based on the expression type of a token. At the time of training the context to the word2vec model, the word2vec model learned four different usage contexts of “vocab”- “vocab” as an assignment, “vocab” as an argument, “vocab” as a subscript, “vocab” as an attribute. Therefore, the first argument of the method calls at line no. 602. **Adding Encoding for Other Programming Language:** In the given example in Figure 4.4,

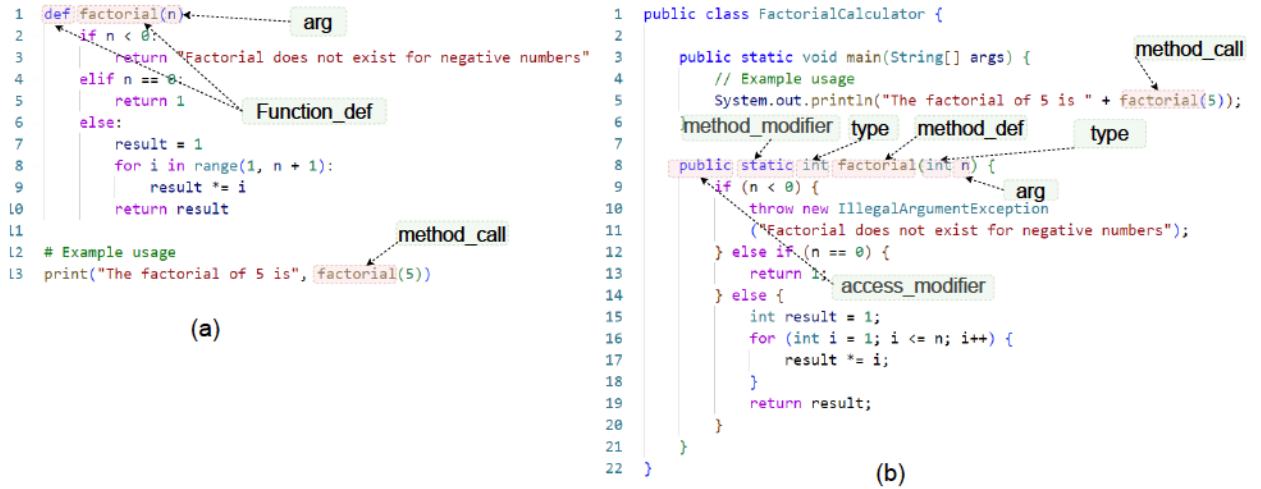


Figure 4.4: Comparison of Encoding For Other Programming Language

segment-(a) shows a simple Python program, and segment-(b) shows a simple Java program, which implies the same as a program at segment-(a). At line 1 in the segment-(a), the encoding of definition is [“def#Function_def,” “factorial#Function_def,” “n#arg”], which implies that the method definition pattern should be (Function_def,Function_def,arg). On the other hand, in line 8 in the segment-(b), we found the encoding could be [“public#access_modifier,” “static#method_modifier,” “int#type,” “factorial#method_def,” “int#type,” “n#arg”]. At line 13 for the segment-(a), “factorial” was encoded as “factorial#method_call” and at line 1 it was “factorial#Function_def”. At line 5 for the segment-(b), “factorial” was encoded as “factorial#method_call” and at line 1, it was “factorial#method_def”; therefore, for the same name “factorial” we can encode the expression type which provides a unique representation of each token, and we believe that this will perform same as Python to another language.

4.5 Experimental Setup

Our dataset consists of 125,496 source files. In total, the dataset contains 24.63 million lines of code. We use the AST module⁶ of Python to parse source files and collect the required information. While the dataset contains 4.18 million method calls, there are 288k unique method calls in them. 21.78% of method calls do not have any arguments, 47.27% of method calls have only one argument, and 30.94% of method calls have more than one argument. For our study, we consider only those method calls that have more than one argument. We use 96,623 files for training and 28,873 files for testing. We use Gensim Word2Vec implementation to detect embeddings⁷. For building the classifiers, we use machine learning

⁶<https://docs.python.org/3/library/ast.html>

⁷<https://radimrehurek.com/gensim/models/word2vec.html>

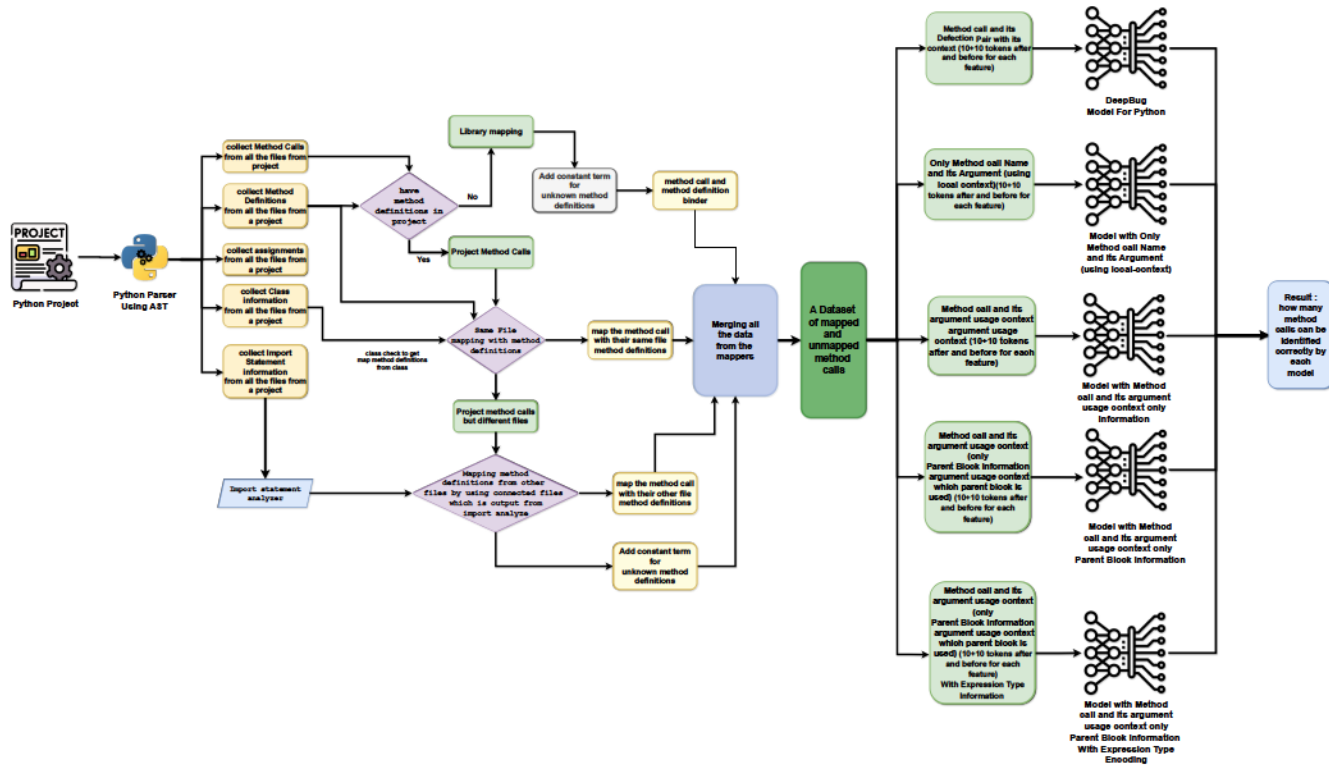


Figure 4.5: Work Flow Diagram

models from the Scikit-learn⁸ library. All experiments are conducted using a machine with an AMD Ryzen 9 5900X processor, 135 GB of memory, and a Geforce GTX 1660Ti GPU.

4.6 Evaluation Procedure

In this section, we will compare AUCMET with DeepBugs and describe our approach’s evaluation process. We also conducted an empirical study to understand the argument expression type and its significance. This research work projects the following research questions:

- **RQ1:** How do programmers use argument and parameter in Python?
- **RQ2:** How effective is our proposed technique (AUCMET) in detecting argument-related bugs compared to DeepBugs?
- **RQ3:** What is the impact of different sources of information?
- **RQ4:** Can we use pre-trained word embedding in detecting argument-related bugs?

⁸<https://scikit-learn.org/stable/>

- **RQ5:** Is our proposed technique efficient enough to be used in practice?

The data and code examples related to the study are available online (on request) to support any future replications.

4.6.1 Evaluation Metrics

We formulate the problem as a binary classification task. To evaluate the performance of the techniques in distinguishing correct code from incorrect one (i.e., bugs created by incorrectly swapping arguments), we consider precision, recall, and F_1 score. Precision refers to the percentage of correct predictions among all predictions. Recall refers to the percentage of correct predictions among all test cases. F1-score refers to the harmonic mean of precision and recall. This is calculated as follows.

$$F_1 \text{ score} = \frac{2 * \text{precision} * \text{recall}}{\text{precision} + \text{recall}}$$

The goal of the classifier is not only to identify incorrect codes but also to detect those codes that are correct. Thus, we reported precision, recall, and F_1 score for both classes for both mapped method calls and all method calls.

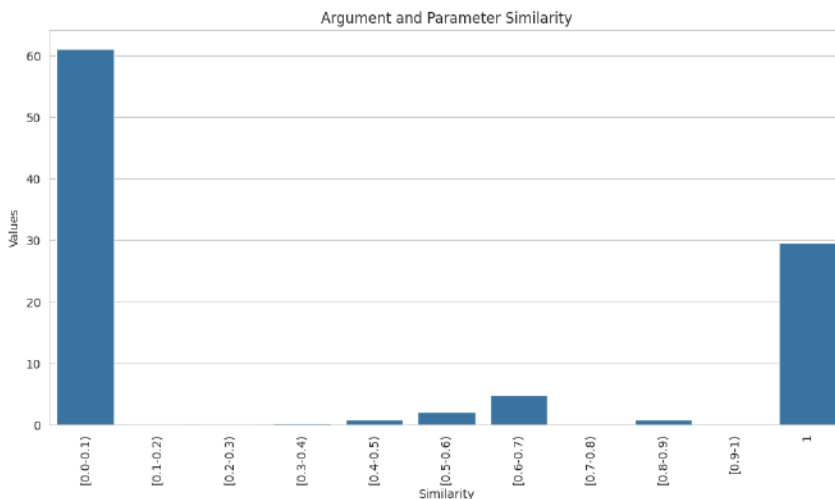


Figure 4.6: Lexical Similarity between Argument and its corresponding Parameter

4.6.2 RQ1: How do programmers use argument and parameter in Python?

We conducted an empirical study on the name-based analysis of arguments and parameters to verify the effect of argument and parameter names. This analysis exposed the similarity between the argument and parameter names, the size of the argument, and the parameter and expression type. Our study is based on the following analysis-

- What is the distribution of lexical similarity between a parameter and its arguments in Python?
- What is the length of arguments and parameters in Python?
- What are the reasons for the dissimilarity of method arguments and their corresponding parameters?
- Can we filter out the arguments that have lower similarity values with their arguments?

4.6.2.1 Distribution of Lexical Similarity between a Parameter and its Arguments in Python?

Motivation:

The declaration statement encapsulates essential information about the method, including its name and parameters. Notably, the parameters and their types are vital in guiding how the method should be called. This linkage between a method call and definition is especially prevalent in modular programming paradigms like Python. If most of the method parameters and arguments lie within a high similarity range, we can conclude that programmers are aware of using the correct arguments and parameters. As an example, in Figure 4.1, the first argument was “vocab,” and the first parameter was also “vocab,” which implied that the developer already knew about the method and its functionality. Therefore, we checked all mapped method calls and their method parameter to check how they use the arguments.

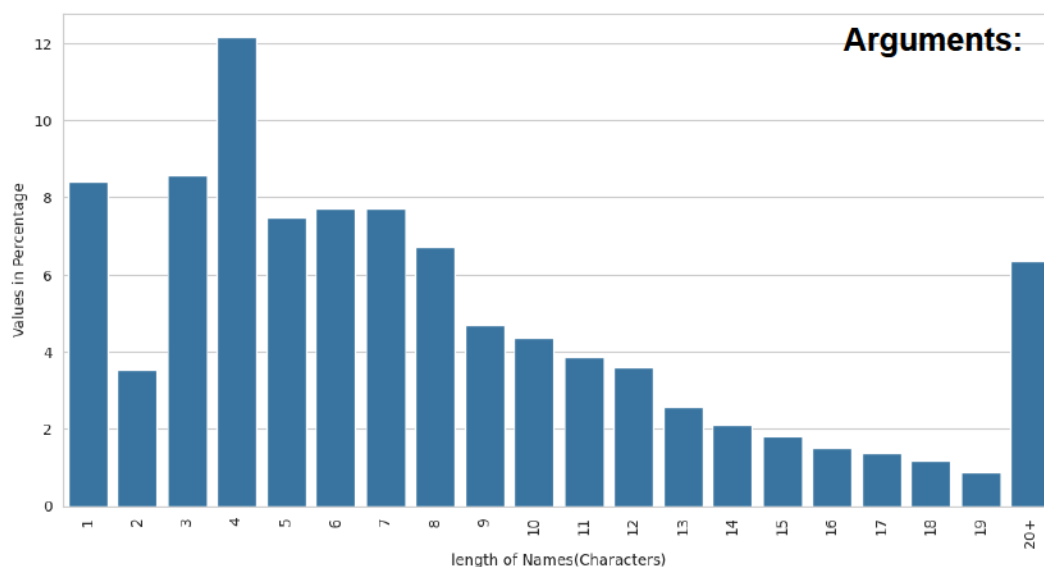


Figure 4.7: Length of Argument Name by Characters

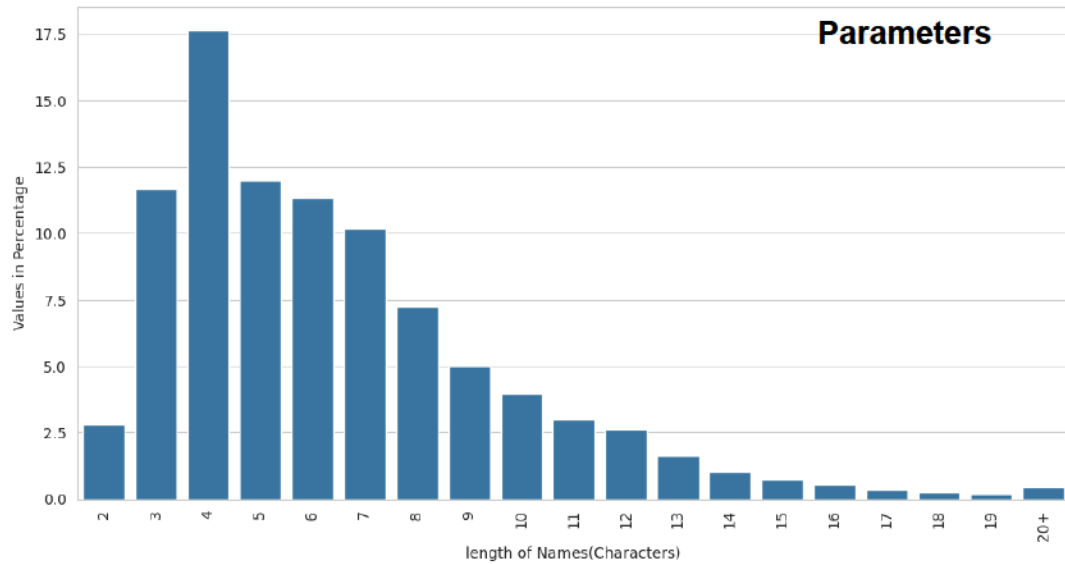


Figure 4.8: Length of Parameter Name by Characters

The similarities between arguments and parameters preserve the correctness of the program structure. Therefore, we intend to detect the name-based similarities so that we can reduce anomalies and get an overview of method calling in Python.

Study Procedure: We collected those method calls from the 150k Python files with method definitions. Thus, we had a pair of arguments and their corresponding parameters. At the time of argument and parameter extraction, we followed the rules to generalize the arguments. (see Table 4.1)

We simplified the complex expressions from our extracted data as complex expressions are independent of name-based analysis for unpredictable usage of names. We used the term similarity concept to determine the lexical similarity of arguments and parameters. Using capital letters and underscores, we tore down the names of method calls and parameters. Finally, we calculated the lexical similarity between an argument and a parameter using the following formula 4.9.

First, to calculate the lexical similarity, we sum up the common terms between argument and parameter and parameter and argument. Then, we determined the total terms in the argument and parameter. After that, we estimated the lexical similarity after dividing the summation of common terms with the total terms of argument and parameter. For illustration,

$$Term_lexi_simi(arg, par) = \frac{|comterms(arg, par)| + |comterms(par, arg)|}{|terms(arg)| + |terms(par)|} \quad (4.9)$$

$Term_lexi_simi(\text{"count"}, \text{"totalCount"}) = (1+1)/(1+2) = 0.67$. Another example is

$\text{lexicalSimilarity}(\text{"totalCount"}, \text{"totalCount"}) = (2+2)/(2+2) = 1$. Here, the value of lexical similarity varies from 0 to 1, where 1 indicates the highest similarity between argument and parameter.

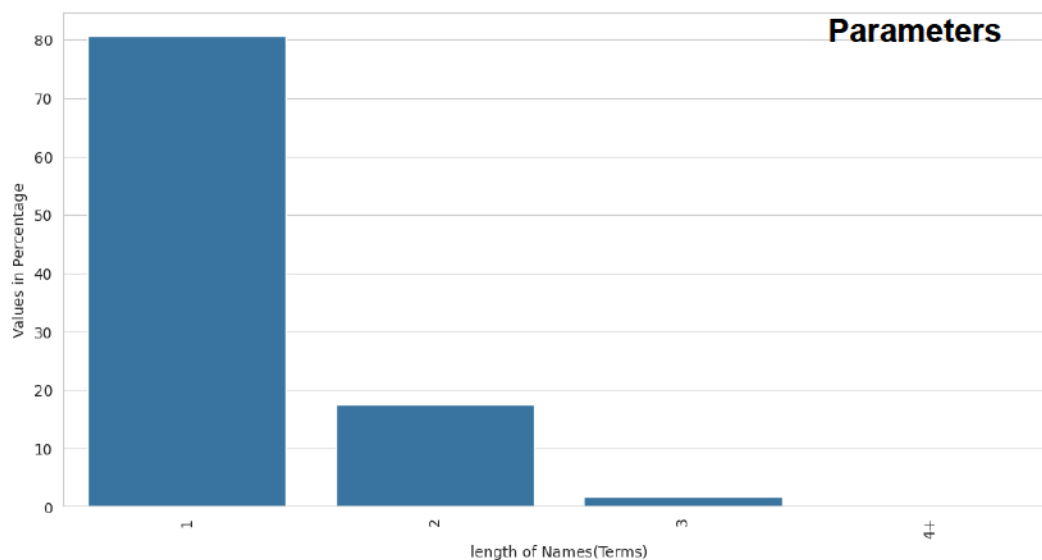


Figure 4.9: Length of Parameter Name by Terms

Result: Our analysis showed in Figure 4.6 that approximately 30% of parameters and arguments had an exact match at their name; on the other hand, more than 60% of arguments and parameters had a similarity range of 0 to 0.1. To find the reason for the low similarity, we analyzed the length of the names of arguments and parameters.

Our goal was to find any relation between the length of the argument and parameters and their similarity. This analysis showed that programmers are not careful about choosing the same name for arguments and parameters. Therefore, a direct approach to creating a model with simple name-based mapping is ineffective for argument recommendation or bug detection.

4.6.2.2 What is the Length of Arguments and Parameters in Python?

Motivation: Upon discovering a consistent distribution of similarities between arguments and parameters, we focus on understanding the underlying reasons for these similarities. Our analysis delved into character and word matches within each word. The primary determinant of similarity lies in comparing the lengths of individual arguments and parameters. Our motivation is rooted in exploring the relationship between length and similarity. This investigation aims to uncover usage patterns and characteristics of method arguments and parameters by examining how their lengths correlate with the observed similarities.

Study Procedure: For our study, we used only the mapped method calls from 150k

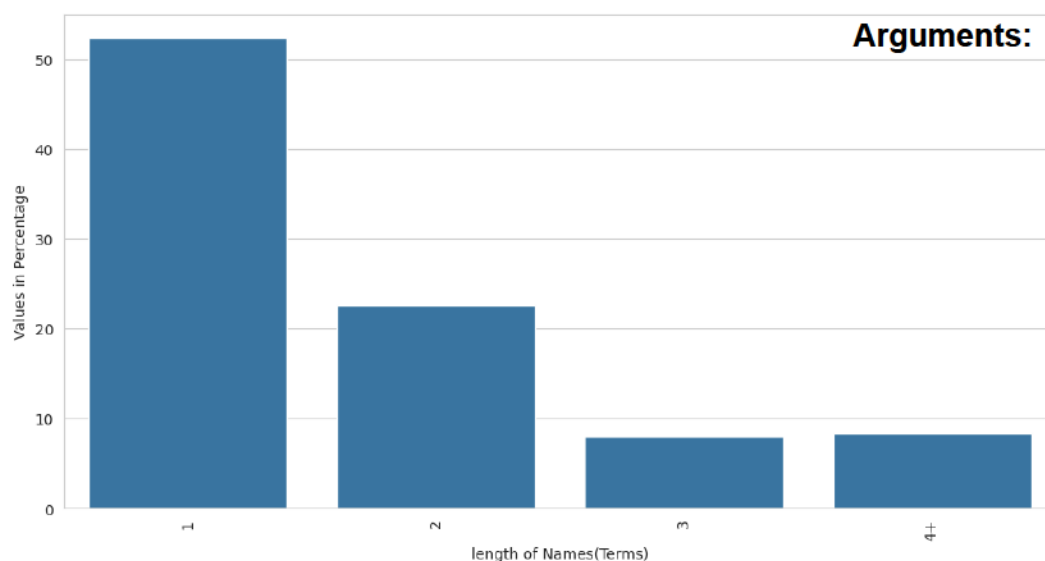


Figure 4.10: Length of Argument Name by Terms

Python files. This length of an argument and parameter depicts how descriptive the names are. As an example, a method call “sum(x,y)” has arguments x and y. A user cannot understand these arguments as they are single characters. If the argument is descriptive, for example, “sum(mark_1, mark_2)”, the user can easily understand that the method call sum will add two marks. Another aspect of collecting the method arguments and parameter length is to show that the lexical similarity of parameters is low while the length is small. We calculated the similarity for all the argument and parameter-mapped method calls for our analysis. We made a cluster of arguments and parameters by their length and generated average similarity values. While considering the term of arguments, Figure 4.10 showed that 50% argument names are only one term, which implies that the names are descriptive, though Figure 4.9 showed the method parameter names are mostly one term (80%). Therefore, we can decide that while choosing the name of an argument, the programmers are not checking the parameter names at all, rather than providing descriptive names considering their programming context and skill for arguments.

Result: Our study shows that the length of the parameter and argument have chances of high similarity. In contrast, the length is longer, and programmers should use descriptive names to reduce errors related to argument selection. The programmers do not provide descriptive names for arguments, and from Figure 4.7, almost 50% names of the arguments are not descriptive (less than 11 characters). On the other hand, more than 80% parameter names are less than 11 characters. Therefore, the programmers are very precise in providing a name for method definitions.

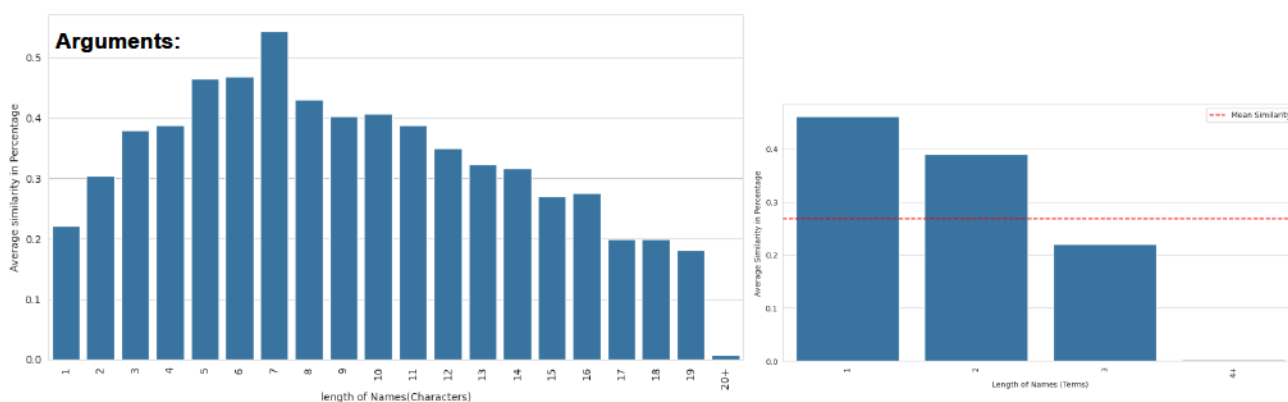


Figure 4.11: Argument Average Similarity by Characters and Terms

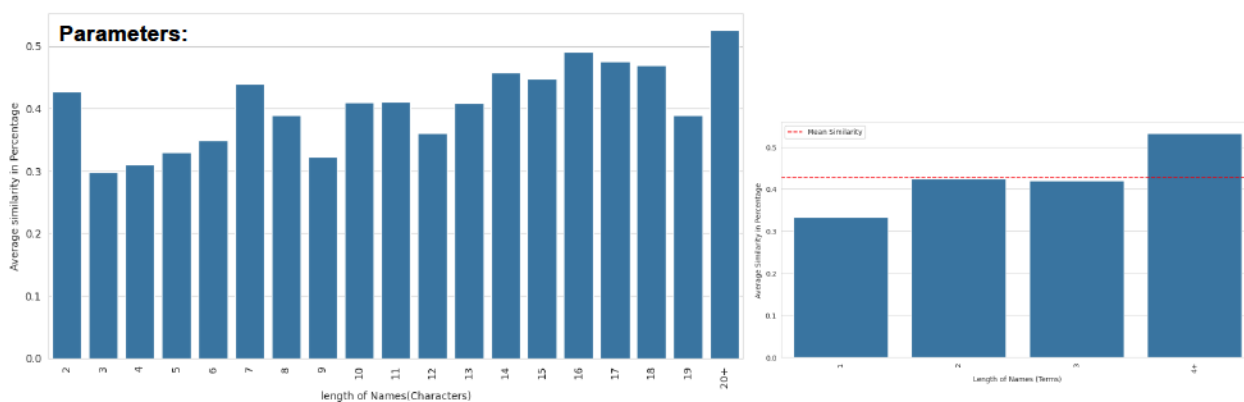


Figure 4.12: Parameter Average Similarity by Characters and Terms

4.6.2.3 What are the reasons for the dissimilarity of method arguments and their corresponding parameters?

Motivation: Our study shows that the arguments of a method call can be lexically different than its parameters. Those dissimilar arguments engender analogies in a software system. Therefore, investigating the reason for dissimilarity will provide the programmer with knowledge about choosing the names of arguments according to the method parameters. Therefore, studying the reason for this dissimilarity is important.

Study Procedure: We filtered out all the dissimilar arguments and calculated the average similarity to determine the reason. We found that most of the arguments have single characters. The figure shows that the similarity value is inversely proportional to the length of the argument and parameter pair. The discrepancy between arguments and parameters is further exacerbated by the use of generic parameter names in methods of collection classes. Approximately 39.6% of the arguments are mapped to parameters with generic names like `index`, `item`, `key`, or `value`. These names are commonly employed in

methods designed to manipulate data collections. Despite the meaningful nature of these parameter names, the corresponding arguments often differ significantly, as they typically represent concrete values or indexes.

Result: We found approximately 75,335 parameters have a shorter name, and they possess a low similarity. This is the reason for the mismatch of parameters and argument usage. In Python, methods are developed and used by others, which indicates that there is no control over the naming of an argument. Though it is unpredictable, it is determined by the length of an argument and its parameters.

4.6.2.4 Can we filter out the arguments that have lower similarity values with their arguments?

Motivation: To develop an argument recommendation system for a given script, the programmer should train a system in such a way that the model learns from the exact names of the arguments. This will be possible when we get similar argument patterns for multiple method calls. Therefore, to build a recommendation system, we first need to collect all of the usage of those arguments and use them in a model for training. For this training process, we need to use a filter that filters out weak and dissimilar arguments from our candidate list. For recommendation, we collected data and treated them as natural language sources. It will be efficient when trained with a list of filtered arguments with higher similarity values.

Study Procedure: We collected lexical similarity of all the parameters and arguments from 150k Python files. We set a threshold value, which is 0.5. We kept those values that have similarity values more than the threshold value. These are the candidate arguments that can be used in the argument recommendation. Therefore, we have a pool of arguments for each method call in the project that a programmer can use for method call completion.

Result: Though we can filter out the arguments with low similarity, we can not build a recommendation system from the filter out candidate. In Figure 4.11 and 4.12, almost 90% argument and parameters had an average similarity of less than 0.5. While using the filter, it removed almost 85% of the candidate arguments. Therefore, using name-based similarity will not be enough to filter out arguments involving low similarity.

4.6.3 RQ2: Effectiveness of the Proposed Technique

Motivation:

We aim to investigate the Performance of the DeepBugs approach and the AUCMET for Python Language. We are also interested in finding the performance of the AUCMET and DeepBugs approach for mapped and unmapped method calls. If the AUCMET performs well for all method calls, it will be verified that swapping argument-related bugs can be detected even if the method definition is missing.

Approach: We divided our dataset into two categories- mapped method call and un-mapped method call (see the section 3.3). The DeepBugs Model depended on both method calls and method definitions. We found that 1,76,743 method calls have a mapping with their method definition, which implies that 4.2% of total method calls can be mapped with the exact project method definition. Among them, if we consider method calls with only definitions in the same file, we get 67,930 method calls with a precise mapping in the same file. We have collected and only mapped the method calls that have definitions in the same project and generated our model to evaluate the performance of DeepBugs for Python. Then, we filled the missing method definition feature with a constant value to use all the method calls for comparison with our proposed technique. Therefore, first, we compared our technique with the DeepBugs technique for those method calls that had exact mapping with their method definitions. Secondly, we compared the performance of our model with the DeepBugs approach for all the method calls from our dataset. Therefore we have built two dataset: Only Mapped Method Call and All Method call.

Result: AUCMET performed better than DeepBugs Model. DeepBugs has a claim for the accuracy of their model (the swapping argument-related bug detectors) for JavaScript Source Code, which is presented in demonstrating a range from 89.06% [2] to 94.70%, whereas for the Python dataset, the same approach has an accuracy of 70.51%. The performance of DeepBugs and our model on Python is shown in Table 4.8. When we used the AUCMET Model for those method calls, which had exact mapping, the accuracy was 89% with an AUC score of 96.80%. Therefore, our model performed better than the DeepBugs Model. In the AUCMET, we did not consider the method definition of a method call and collected code context from the previously used code token, which implies that it is possible to generate argument-swapping related bugs from the code context. For the second comparison of our proposed technique (AUCMET) with the DeepBugs Approach, we trained both of the models with all the method calls, and the performance is included in Table 4.9. Besides, in Table 4.8, it was shown that for the prediction of correct sequence of arguments, DeepBugs showed up with poor performance of 68% precision, whereas our

Table 4.8: Performance Comparison with DeepBugs Approach and AUCMET For Mapped Method Calls

Performance Metrics	DeepBugs model		AUCMET	
	Wrong	Correct	Wrong	Correct
Accuracy	70.51		89	
precision	77	68	85	94
recall	62	82	95	83
f1-score	69	75	89	88
AUC	81.59		96.80	

Table 4.9: Performance Comparison with DeepBugs Approach and AUCMET For All Method Calls

Performance Metrics	DeepBugs model		AUCMET	
	Wrong	Correct	Wrong	Correct
Accuracy	69.8		90.79	
precision	69	73	94	88
recall	76	65	87	95
f1-score	72	69	91	91
AUC	79.73		97	

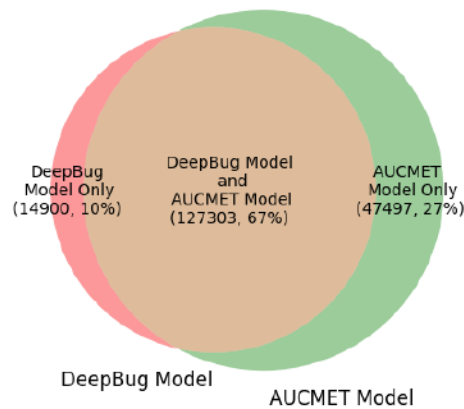


Figure 4.13: Performance of Both DeepBugs Model and AUCMET

proposed technique showed a precision of 94%. For all the method calls, our dataset size is 19,51,736, which is 11 times larger than the dataset of mapped method calls. On the other hand, for all of the method calls in our dataset, the DeepBugs approach for Python lost its accuracy due to a lack of method definition information. Besides, in Figure 4.13, we showed where the AUCMET performed better than the DeepBugs Model and successfully identified 47,497 cases, which is three times more than the correct prediction of the DeepBugs Model. Surprisingly, our model performed almost similarly even though we trained our model with copious amounts of vocabulary or tokens. Table 4.9 showed the effectiveness of our approach, covered the lack of method definitions and issues of dynamic type binding of method arguments, and supported the large-scale study of source code context.

4.6.4 RQ3: Impact of Different Source of Information

Motivation: Our proposed technique considered three features of a method call and its arguments. We considered different combinations of these features to find the impact of the source of features. Sources of information implied the source of the context of a feature

Table 4.10: Combination Of Different Source Information Of AUCMET

Model	Model Configuration
1	MC _{A1} _LC +MC _{A2} _LC
2	MC _N _LC+MC _{A1} _LC +MC _{A2} _LC
3	MC _N _LC+MC _{A1} _LC +MC _{A2} _LC+MC _{A1} _Type+MC _{A2} _Type
4	MC _N _LC+MC _{A1} _AUC +MC _{A2} _AUC+MC _{A1} _Type+MC _{A2} _Type
5	MC _N _LC+MC _{A1} _PCLU +MC _{A2} _PCLU+MC _{A1} _Type+MC _{A2} _Type
6	MC _N _LC+MC _{A1} _LC +MC _{A2} _LC+MC _{A1} _AUC +MC _{A2} _AUC +MC _{A1} _PCLU +MC _{A2} _PCLU+ MC _{A1} _Type+MC _{A2} _Type
7	MC _N _LC_ET+MC _{A1} _LC_ET +MC _{A2} _LC_ET+MC _{A1} _Type+MC _{A2} _Type
8	MC _N _LC_ET+MC _{A1} _AUC_ET +MC _{A2} _AUC_ET+MC _{A1} _Type+MC _{A2} _Type
9	MC _N _LC_ET+MC _{A1} _PCLU_ET +MC _{A2} _PCLU_ET+MC _{A1} _Type+MC _{A2} _Type
10	MC _N _LC_ET+MC _{A1} _LC_ET +MC _{A2} _LC_ET+MC _{A1} _AUC_ET +MC _{A2} _AUC_ET+ MC _{A1} _Type+MC _{A2} _Type
11	MC _N _LC_ET+MC _{A1} _LC_ET +MC _{A2} _LC_ET+MC _{A1} _AUC_ET + MC _{A1} _PCLU_ET+MC _{A2} _PCLU_ET+MC _{A2} _AUC_ET
12	MC _N _LC_ET+MC _{A1} _LC_ET +MC _{A2} _LC_ET+MC _{A1} _AUC_ET + MC _{A1} _PCLU_ET+MC _{A2} _PCLU_ET+MC _{A2} _AUC_ET+ MC _{A1} _Type+MC _{A2} _Type

(as an example- line number, tokens from that line). Our goal is to find the impact of the context of these features. We collected contexts for the features (described in the section 4.2). Therefore, we had local Context, Parent Context, Argument Usage context for Argument, and Method call. We checked 12 different combinations of models to investigate the performance of the AUCMET. Table 4.10 showed the model description (for abbreviation of notation at Table 4.10 and see the Tables 4.4,Table 4.5 for context information). We trained 12 different models with the same training examples and tested with the same test examples.

Study Procedure: Our first model combined the Local Context of first Argument of the Method Call and the Local Context of second Argument of the Method Call. This model showed an accuracy 4.11 of 62% with a recall of 80%(for wrong sequence) and 44% (for correct sequence). This model gained information from the name-based argument local context. However, from this model, we can only understand how the arguments should be sequenced, though we did not get any idea with which method the arguments were associated.

In our second model we added method call information with the first model, and we intended to take method call information (local context of method call) to train the model about which method argument was related to which method call. The accuracy of this model was not increased significantly where the recall was 77% (for wrong sequence) and 50% (for correct sequence) 4.11. Therefore, the model started to learn the correct pattern more accurately than the first model.

Table 4.11: Analysis of Different Sources of Information

Model	Performance Metrics					
	Class	Accuracy	precision	recall	f1-score	AUC
1	Wrong	62	59	80	68	69.08
	Correct		69	44	54	
2	Wrong	64	61	77	68	71.4
	Correct		68	50	58	
3	Wrong	70.42	67	81	73	79.67
	Correct		76	60	67	
4	Wrong	68	74	56	64	76.43
	Correct		65	80	71	
5	Wrong	67	65	74	69	74.61
	Correct		70	59	64	
6	Wrong	77.94	75	84	79	87.87
	Correct		82	71	76	
7	Wrong	74	82	60	70	74.61
	Correct		69	87	77	
8	Wrong	85	90	80	85	94
	Correct		82	91	86	
9	Wrong	78	86	67	75	86.98
	Correct		66	54	59	
10	Wrong	81	75	93	83	91.85
	Correct		90	70	79	
11	Wrong	89.19	89	90	89	89
	Correct		90	89	89	
12	Wrong	90	87	94	90	96.94
	Correct		94	85	89	

In our third model, we introduced the type of information from the local context of the method arguments; this information is important because, for a position, a specific type of method argument should be used. Therefore, the model could distinguish the pattern of an argument by its type. In Table 4.11, the third model had an accuracy of 70.42%, and the recall value increased from 77% to 81% for wrong argument sequence and from 50% to 60% for correct argument sequence. Therefore, this model started performing better when the argument was uniquely defined by its argument type.

We did not use the local context for the fourth and fifth models. Instead, we tried only argument usage context. We collected two different usage contexts: Parent Usage Context (PC) and Argument Usage Context(AUC) (see section 4.2). The fourth model considered only AUC with the type of the argument, and the accuracy dropped to 68%, and both recall and precision for both classes went down compared to the previous models. The fifth model considered the PC, which also showed a comparatively lower range performance. Therefore, while the model missed the local context, its performance decreased. This is because an

argument can be changed on any line in the source code, and for a certain name, it may get the context multiple times. Therefore, the model was not learning perfectly when it did not get the local context.

Learning from the result of the first five models, we trained our new model (Model 6) with all the context from the features. Thus, model 6 first learned from the local context to identify the usage at the location it used; then it looked up how it was used previously from the AUC (see section 4.2) information, and as the last context, we added the parent information, which limited the scope of the usage of the argument. From Table 4.11, Model 6 performed better than others for the above reasons. The accuracy for the unseen dataset was 77.94%, and recall was 84% for the wrong sequence and 71% for the correct sequences. Therefore, this could be our proposed technique with an AUC value of 87.87%.

When we went through example after example, we found our model performed well for those unique examples, and the usage patterns differed. For example, a token was used in five times in a source code before using it in as an argument method. Though the usage contexts could be different, they were not always obvious. Therefore, the above-mentioned models face ambiguity issues with the same name and context. Our source code was encoded with their expression type information and generated the last six models at Table 4.10. The model 7 at Table 4.10 was the extended version of the model 3, and the accuracy increased from 70.42% to 74%. The reason, hidden behind the increment of accuracy, was that the extended version of context carried the structural information, which taught the model about the uniqueness of each token. Similarly, model 8 and model 9 are the extended versions (added expression type) of model 4 and model 5, respectively. This model improved their accuracy for both classes and worked perfectly fine. We built another model with the local context and the Argument usage context with the type information of the method calls and their arguments. This model was denoted as model 10, and it performed better than other models with a 93% recall value for wrong sequence and a 93% recall value for correct sequence and showed 81% accuracy (see Table 5.4).

Finally, we checked the performance of two models with all the context with their extended-expression information. We also checked whether the argument type had an impact on the performance or not. Our model 11 was built on the same concept as model 6 except for the type information, and the final model used all the context with expression type embedded with the context. Table 5.4 showed the performance of those two models. The model 11 showed an accuracy of 89.19% whereas the model 12 showed an accuracy of 90%. If we consider the recall for model 12 for the wrong sequence was 94% whereas model 11 showed 90%. Therefore, model 12 is our final model and is named the AUCMET .

Result:The AUCMET and model 11 performed better for detecting correct and wrong sequence.

4.6.5 RQ4:Performance Comparison Of AUCMET With Pre-Trained CodeBert

Motivation: Pre-trained models are trained with huge and diverse datasets, and they can be reused multiple times for machine learning and deep learning works. A pre-trained model is already balanced with weight and bias, which can handle any similar real-time data. CodeBert is a masked language model that can generate a token from a given context. Therefore, they can be used to generate a vector, which can be used to train another deep-learning model. This supports bimodal(which considers the relation between document and code) and unimodal (which considers either code or documentation).

Table 4.12: Performance Comparison of DeepBugs, AUCMET, Pre-trained Large Language Model-BERT

Performance Metrics	DeepBugs model		AUCMET Model		Pre-trained Large Language Model-BERT	
	Wrong	Correct	Wrong	Correct	Wrong	Correct
Class						
Accuracy	69.8		90.79		71	
precision	69	73	94	88	72	70
recall	76	65	87	95	69	73
f1-score	72	69	91	91	71	71
AUC	79.73		97		80.74	

The chosen hyper-parameters for CodeBert training included a batch size of 2,048 and a learning rate of 0.0005. Parameter updates were performed using the Adam optimizer, with the number of warmup steps set to 10,000. The maximum sequence length was capped at 512, and the training was designed to run for a maximum of 100,000 steps. It performed better than other pretrained model because a combination of carefully chosen hyperparameters, the use of advanced optimization techniques like the Adam optimizer, and strategic decisions regarding the learning rate, batch size, and dataset configuration. Our goal was to compare our AUCMET and pre-trained Model-CodeBert. We wanted to check the importance of our collected context. If our AUCMET performed better than the model that considered the information trained from an unknown source code context, it would be verified that the local context from a certain project is better than the unknown source code context. On the other hand, the model performs faster than ordinary word2vec. Therefore, if the accuracy of the generated model from the pre-trained vector is higher than our model, our vector generation from the code context will be pointless.

Study Procedure: We used the same extracted data for this analysis. However, we generated a new vector from a large language model. We used the CodeBERT model [47] [48] to generate this vector. The CodeBERT model provides a vector from the context it trained

from for a word. Therefore, we used the same process to build correct and wrong code examples and the same model for training and testing. The accuracy for the trained model from generating vector by the Pre-trained model⁹ is included here in Table 4.12. For context collection, we used the binder information from Table 4.2, but we used only the name as a query of the Pre-Trained Model. After getting the vector from the pre-trained Model, we used the same sequence generator to generate the wrong examples for our new model.

Result: When comparing the three models, the AUCMET outperforms the other two across all metrics. It has the highest accuracy, which indicates overall effectiveness in classification, and the highest precision and recall. The AUCMET is excellent at identifying true positives while minimizing false positives and negatives. The AUCMET has an AUC value of 97, which supports the ability to distinguish between classes effectively across various threshold settings. On the other hand, we found a 20% increment in performance for each metric of the AUCMET to the pre-trained-embedding-based Model.

The Pre-trained Model-CodeBERT and the DeepBugs Model have similar performance levels, with BERT having a slight edge in precision and AUC but lower recall than DeepBugs. However, both are significantly less effective than the AUCMET.

4.6.6 RQ5: Efficiency of the Proposed Technique

Our approach involves extracting features, mapping the method call and its definitions, context collection and context preprocessing, vector generation of features, training, and testing of four different models. Our most time-consuming module is vector generation, as we have generated vectors for 9 different features for our proposed model for each example, and it took 5.9 days to generate vectors for our neural model. For each example, the features Extraction took 13 ms, and the mapping of the method call and its definitions occupied 15-20 ms on average (based on the number of files in a project). Context collection and context preprocessing consumed 13 ms per example, which is comparatively faster as we have used one-time tokenization of the project. Within 97 minutes, training was taken on our machine, which is comparatively faster. Testing of our model took 3 mins in total for 497 projects of 1,76,302 examples. Therefore, our proposed model can generate a warning of swapping argument within 1.1 ms.

4.6.7 Additional Analysis To Evaluate The Performance of AUCMET

Our AUCMET performed better than other models. We checked the performance of the AUCMET from the following three perspectives and showed where and why the AUCMET failed to perform better than the DeepBugs Model.

⁹https://huggingface.co/docs/transformers/en/model_doc/bert

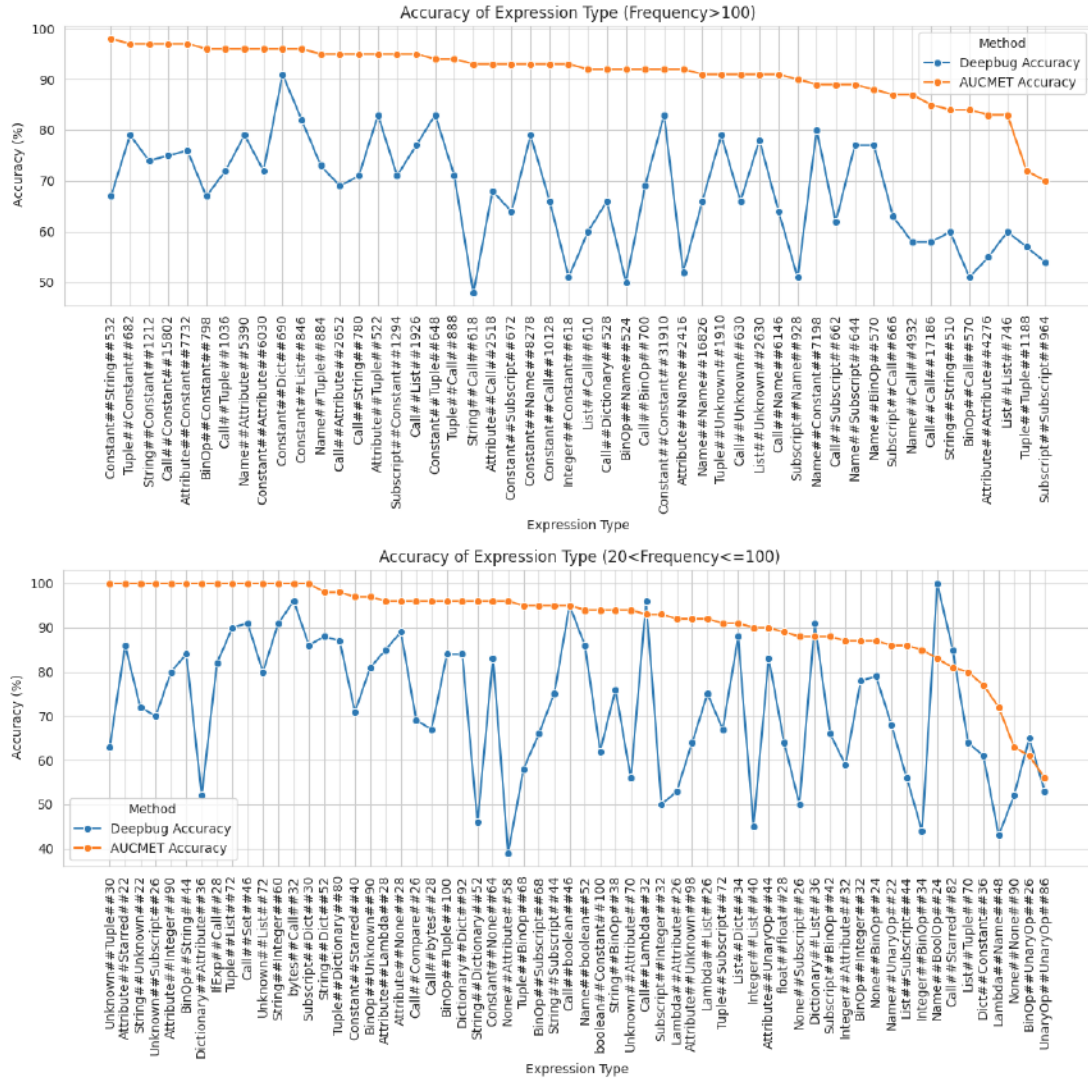


Figure 4.14: Accuracy of Expression Type-based Analysis ($20 \leq \text{Frequency}$)

4.6.7.1 Performance of DeepBugs on Python and AUCMET based on Expression Type:

Both DeepBugs on Python and the AUCMET considered two subsequent arguments. We found 326 combinations. Among them, 50 types are the most common (more than 100 frequencies, for example). In the Figure 4.14 in the first figure, for all of the categories, the performance of DeepBugs is less than our AUCMET, and in the second figure, only “call+lambda” which has only 32 examples of “Dictionary + List” with 36 examples, “Name + boolean” with 24 examples and “BinOp+ UnaryOp” with 26 examples performed better than our model. Again, in Figure 4.15, “Name + IfExpr”, “Call+Boolor”, “Call+ListComp”, “List + UnaryOp”, and “unknown+ Dictionary” performed slightly

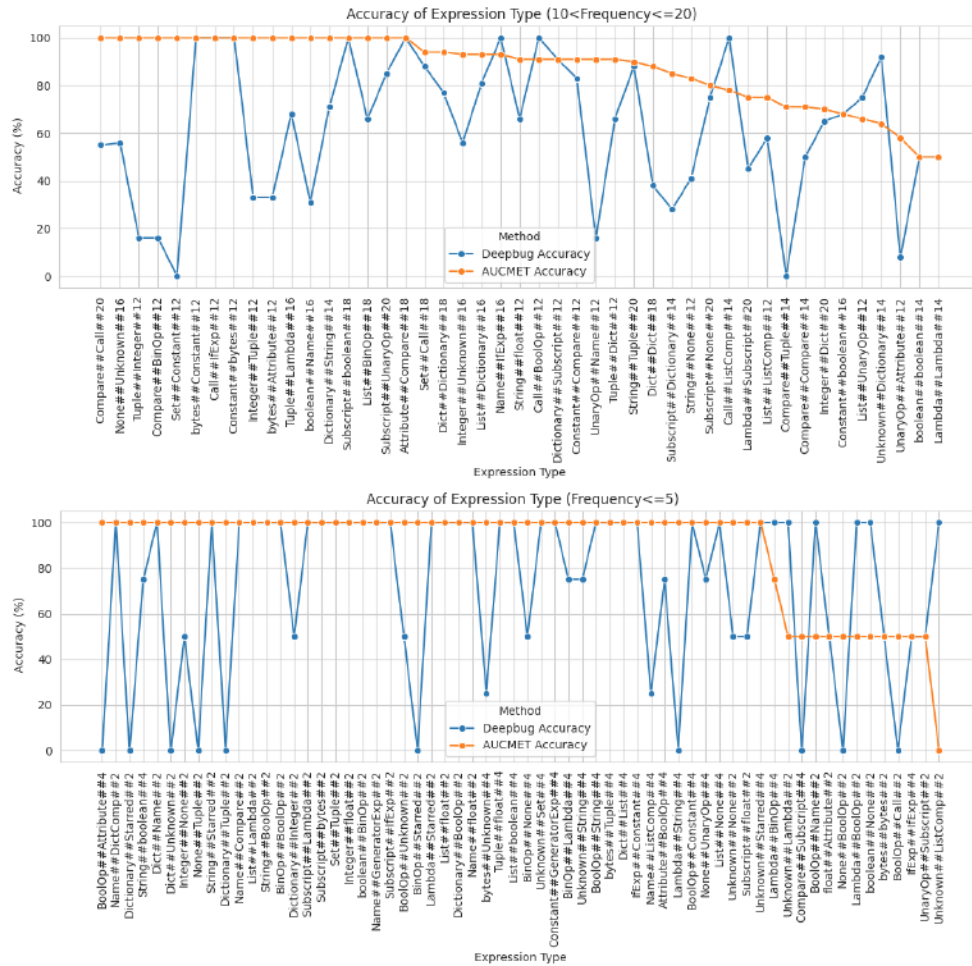


Figure 4.15: Accuracy of Expression Type-based Analysis ($20 \leq \text{Frequency}$)

higher than our model. Among the 1,96,368 examples, 192 of our AUCMET performed slightly lower than DeepBugs. On the other hand, for 1,96,176 method calls, the AUCMET performed on average 20% higher than the DeepBugs model. These 0.09% examples are unique and negligible in this context.

4.6.7.2 Performance of DeepBugs on Python and AUCMET based on Context Length

Motivation: To check the performance of the AUCMET and DeepBugs by the context, we checked the performance of cases in both models that could successfully catch and put the average context length with that. Therefore, we aim to check whether the long context is needed for better training.

Study Procedure: We used our final model, discussed at 4.9, and took the result for those test cases. Then, we collected all the context lengths with respect to individual

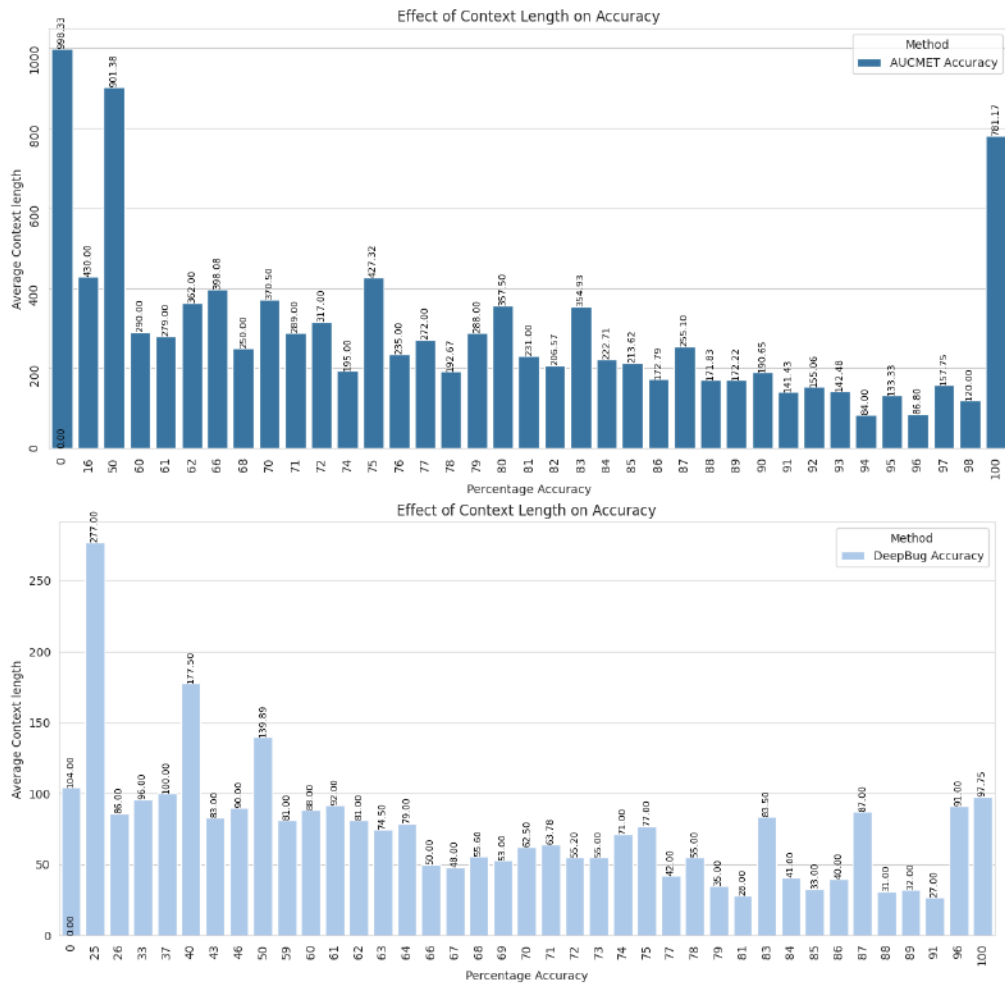


Figure 4.16: Effect of Context Length on Performance For DeepBugs and AUCMET

accuracy. We wanted to conclude the range of average length for which the result was better.

Result: In Figure 4.16, we plotted accuracy against the average context length. We found that accuracy was high for the lower context length. The AUCMET performed well when the context length was below 400 tokens. However, we found an accurate performance for 781.17 average tokens. We checked the number of examples for those test cases, which was 37882. On the other hand, for the DeepBugs model, the average context length was less than 150 tokens. If we compare the accuracy range with our model and the DeepBugs model from 50% to 100%, the average context length will be less than 400 for The AUCMET and less than 100 for the DeepBugs model. Therefore, a sound-long context of 400 tokens that are related to the method call and its argument will perform better for swapping argument-related bugs.

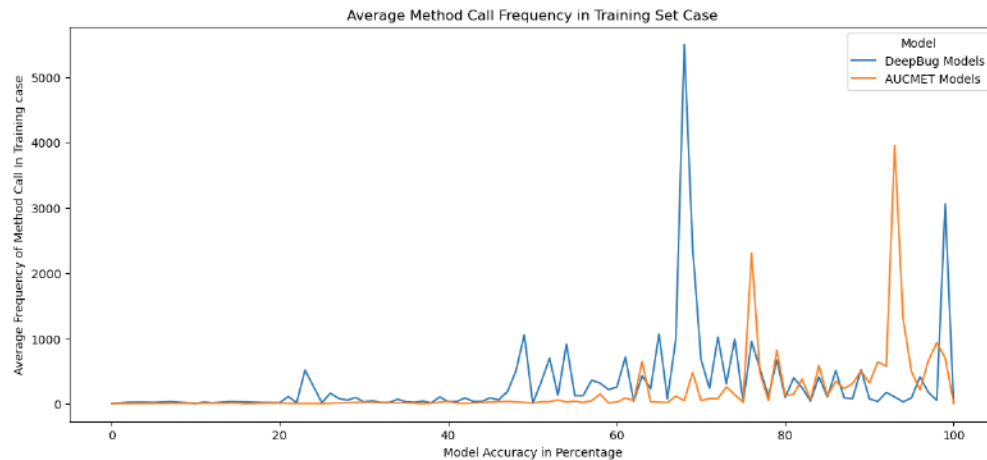


Figure 4.17: Average Method Call Frequency in Training Set Case

4.6.7.3 Performance of DeepBugs on Python and AUCMET based on Method Call Appearance in Training Examples

Motivation : We wanted to check the performance of the AUCMET and DeepBugs Model with respect to the appearance of method calls in the training segment. All of the method calls in the training set are contextually unique; our goal is to check whether the method call's appearance controls our model's performance.

Study Procedure: We collected the number of unique method calls from the training set and collected the frequency of each method call. Our goal is to find the appearance of examples in the training and test cases and check if there is any impact on the accuracy of the frequency of method calls in the training set. We collected all the unique method calls from the training and test examples. Then, we matched the method call name from the test cases and matched it with the examples. After getting the matched examples, we divided the dataset into two parts- the accuracy of the AUCMET with the appearance of the method call and the accuracy of the DeepBugs Model with the appearance of the method call. Then, we plotted the graph by using the average count of appearance of the method, which calls for accuracy and accuracy. If the accuracy is 95% we collected all the appearance counts of that method call, which are showing 95% accuracy. We followed the same approach to collect the average count of Appearance in the training set for the DeepBugs Model.

Result: Figure 4.17 shows the performance based on the average approaches of method cal in the Training example. For both of the models, the accuracy is poor for both those method calls that appeared less than 3000. On the other hand, it shows that it has 80% accuracy even if it has less appearance of method call in the training set. **Therefore, the appearance frequency did not have any impact on the accuracy. The context of**

a method call carries significant importance for bug detection.

4.7 Threats to Validity

This section discusses threats to the validity of this study. Even though we covered the research gap of large-scale study and language dependencies, to remove the complexity of method call argument-related bug detection, we simplified the arguments in which expressions are arbitrary and complex.

First, we consider a dataset consisting of 150,000 source code files written in Python language. One can argue that the results we obtain may not generalize to other Python projects or projects written in other languages. However, we would like to point to the fact that we consider a large number of Python files that were collected from a large number of projects. Thus, our results should largely carry forward.

Second, for the purpose of this study, we map method calls to their definitions. Method calls, and their definitions are not necessarily in the same file. Incorrect mapping can introduce bias in our findings. We investigated for a significant time to ensure the correctness of the program. A manual investigation of randomly sampled 8000 method calls and their definitions showed that our implementation can accurately map 85% of method calls. For the remaining method calls, we received multiple mapping to method definitions. Thus, we ignore those method calls from our analysis.

The performance of our technique depends on the word embedding technique and the setting used to run the algorithm in this study. To simplify the implementation, we consider the Word2Vec model, a popular word embedding technique used in many other prior studies. One can argue that the model(s) performance can be affected by the selection of the algorithm. We did not focus in this direction because our goal is to evaluate the effectiveness of considering argument usage patterns. Thus, we did not consider that in this study. Future research should focus in this direction.

Third, we only consider the lines before the target method call to collect the context of the argument usage. However, it is also possible to consider the bottom lines. However, the long code dependency on a token in a program was not considered here; this remains a work in the future.

4.8 Conclusion

Our study investigates the effectiveness of the DeepBugs approach for Python. On the other hand, it shows if the approach can be used for any dynamic programming language. Our study showed that this approach can be replicated for other programming languages, but the structural changes from programming language to language changed the result.

Therefore, we collected our features to cover all the implications for Python and showed that usage context-based analysis of any token is effective for bug detection or anomaly detection in Python. Our proposed dataset contains the mapping information using our napping algorithm, which improves data collection. Besides, we solved the programming vocabulary ambiguity, which shows a 10% better result than previous approaches. Therefore we can concluded that it is not mandatory to map the method call to its definitions, the usage pattern of a variable can be used to determine correct location of variable usage and for very large scale study context length decreased the accuracy of bug detectors which was solved by inducing structural information. This increased the accuracy from 69.8% to 90.79%. The performance of AUCMET was the same for both mapped and unmapped method calls. Therefore, for the smaller dataset (mapped method calls, which is 10% of the whole dataset) , the AUCMET model showed an accuracy of 89%, and for all method calls, it showed 90.89%, which removes the issue of a large-scale study.

Chapter 5

An Empirical Study of Argument Recommendation by LLM in Python

5.1 Introduction

Argument recommendation is one of the most important features of automatic code completion in Integrated Development Environments (IDEs), which is a part of Code completion [34] [49] [50] [51]. This feature can help the developer generate the next argument from the code context. These Argument recommendation techniques are mostly based on the context or previously written lines of code where the arguments are declared or used, which is considered a static property in the source code. This has already been used for source code modeling [34] [52] [49], code summarization [53] [54], code clone detection [55] [56] [33], and program repair. Large language models like GPT-4, T5, LLAMA2, and so on are becoming very popular for solving software engineering-related problems such as code completion or code summarization. To the best of our knowledge, the capability of this **pre-trained LLM models** to complete the argument list with an API call based on the current code context has not yet been evaluated.

In this study, we aim to evaluate the efficacy of Large Language Models (LLMs) in suggesting appropriate arguments for API calls during software development. These pre-trained transformer models are trained with large-scale source code infractions and balanced with **learned biases and weights**. Therefore, these models should be able to recommend argument/s for an API call based on the given context. Training with our dataset will consume time (see section 4.6.6) for generating code tokens, and balancing those weights and biases is challenging. Therefore, our study aims to investigate the performance of three popular Pre-trained models, CodeBERT and Code Llama, for argument recommendation

in Python.

5.2 Background

5.2.1 Statistical Language Models

Large language models [52] are generally used to generate a sequence that depends on another sequence. A source code token is directly related to the other sequences in the same source code. Therefore, the probability of generating tokens from a given source code is a statistical machine translation or source code generation task. Therefore, tokens with possible candidates can perform better in this context from a given sequence and source code. A language model compares a token sequence with its existing token sequence (the processed tokens used for training the Model) to generate the nearest possible tokens. Given a sequence of S_t , a Statistical language model for source code will provide a possible list, $P(S_t)$ which is determined by 5.1.

$$P(S_t) = \sum_{k=1}^n P(sw_k | sw_{k-1}) \quad (5.1)$$

In the equation 5.1, the sw implies the token sequence, and n is the number of tokens. As the predicted token depends on the existing previous token, the statistical model calculates the probability of the appearance of the next token from the previous tokens.

The figure displays two side-by-side screenshots of code editors showing code completion for a Python file named 'dispatch.py'. The code defines a 'Signal' class and its methods. The left screenshot is from Visual Studio Code, and the right is from PyCharm IDE. Both show the same code with different completion suggestions.

```

1 import weakref
2 from dispatch import saferef
3 WEAKREF_TYPES = (weakref.ReferenceType, saferef.BoundMethodWeakref)
4 def _make_id(target):
5     if hasattr(target, 'in_func'):
6         return (id(target.in_self), id(target.in_func))
7     return id(target)
8
9 class Signal(object):
10     """
11     _debugging = False
12     def __init__(self, providing_args=None):...
13     def connect(self, receiver, sender=None, weak=True, dispatch_uid=None):...
14     def disconnect(self, receiver=None, sender=None, weak=True, dispatch_uid=None):...
15     def send(self, sender, **named):...
16     def send_robust(self, sender, **named):...
17     def _live_receivers(self, senderkey):...
18     """
19     none_senderkey = _make_id(None)
20     receivers = []
21     for (receiverkey, r_senderkey), receiver in self.receivers:
22         if r_senderkey == none_senderkey or r_senderkey == senderkey:
23             if isinstance(
24
25
26

```

Visual Studio Code completion suggestions (left):

- Warning
- and
- assert
- async
- await
- break
- case
- class
- continue
- def
- del
- elif

PyCharm IDE completion suggestions (right):

- receiver
- None
- self
- receivers
- receiverkey
- lambda
- not
- len(__obj)
- print(values, sep, end, fil...
- Exception
- sum(__iterable)
- if r_senderkey ==
- if isinstance(

Figure 5.1: An Example of Code Completion in Visual Studio Code and PyCharm IDE

5.2.2 Code Completion in Python

Automatic code completion became a common feature in modern IDEs. It is also integrated into most of the text editors. For the generation of the code completion model, a large number of information must be used for training. Training is required to learn multiple token sequences and programming patterns. When completing any partial code segment, IDE usually suggests the name of variables, API name, Import statements, or any associated code segments. In Figure 5.1, we showed a simple code completion technique integrated into Visual Studio Code and PyCharm. At the backend of those, IDE uses large language models, and for PyCharm, the model involved for code completion was trained with 100 million parameters and a context length of 1,536 tokens¹, which is significantly large. Therefore, code completion tasks can be done with the help of a Large language model. The code completion can be done by a frequency-based approach or an alphabetical approach. We are highly motivated to investigate the performance of a large language model for argument completion from a given context.

5.2.3 Argument Recommendation

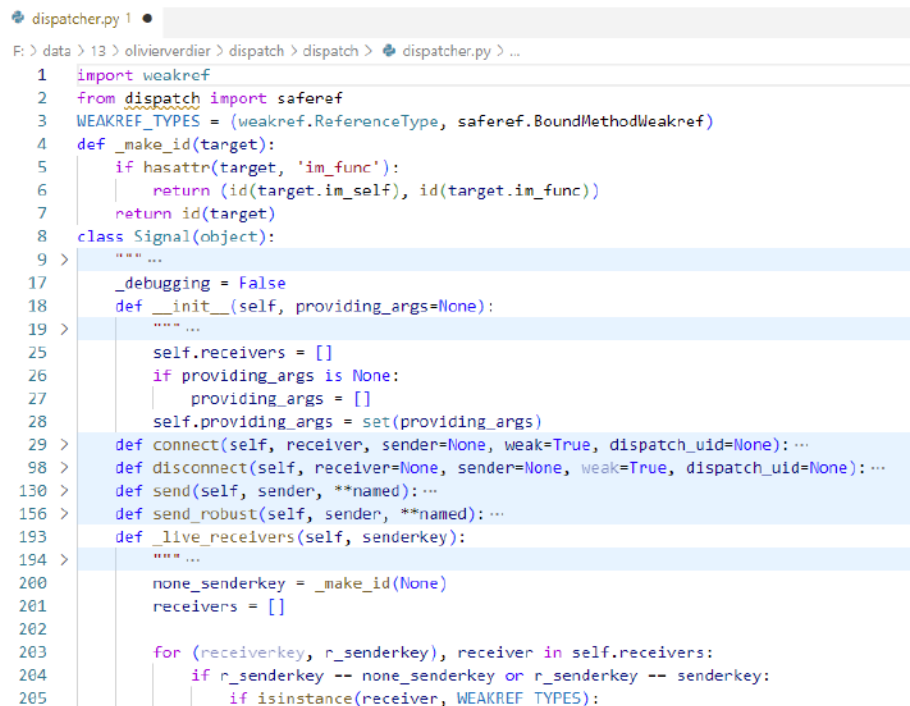
A correct argument passing supports the correctness of a program. For context-aware code completion techniques, it is common to fill up the sequence from a given sequence. For argument recommendation for a method call, IDE will list possible candidates of argument from the list of variables. There can be multiple suggestions for filling up the argument. A rule-based filter is used to reduce the candidate list, as this filter will eliminate unnecessary variables from the list. A similarity-based approach [17], statistical model-based approach [52], and determination of localness of API arguments [57] are the traditional approaches for argument recognition. Code Llama provides a model for generating Python sequences and provides four different models with 7B, 13B, 34B, and 70B parameters as source code is a long context-sensitive source of natural language. It was trained with a long operating sequence of context (up to 16384 tokens) and used a quadratic complexity of attention for balancing the short to long contexts. This code LLAMA can be used to generate a sequence from a given sequence that can be used for argument list generation.

5.2.4 Usage of Large Language Model

Existing popular language models are CodeT5, CodeBERT, and Code Llama, which are used to generate code sequences. CodeT5 is an encoder-decoder model based on the T5 architecture [58]. This model can suggest identifier names with code and comments generation simultaneously. We suggest a new way to train models that consider identifiers,

¹<https://blog.jetbrains.com/blog/2024/04/04/full-line-code-completion-in-jetbrains-ides-all-you-need-to-know/>

aiming to improve the alignment between natural language descriptions and programming language code. We introduce a dual-generation task that simultaneously generates code and comments. Our model can handle both understanding and generating code, supporting multi-task learning. CodeBERT is a bidirectional Transformer model that is trained using a multilayer Transformer model. The purpose of the model is to learn better from a large number of unimodal codes and generate code sequences from a new context. Python is one of the six programming languages, and the trained model can be fine-tuned to a new context. This model can be used for code search and code-to-text generation tasks; we are



```

dispatcher.py 1
F: > data > 13 > olivierverdier > dispatch > dispatch > dispatcher.py > ...
1 import weakref
2 from dispatch import saferef
3 WEAKREF_TYPES = (weakref.ReferenceType, saferef.BoundMethodWeakref)
4 def _make_id(target):
5     if hasattr(target, 'im_func'):
6         return (id(target.im_self), id(target.im_func))
7     return id(target)
8 class Signal(object):
9     """ ...
17     _debugging = False
18     def __init__(self, providing_args=None):
19         """ ...
25         self.receivers = []
26         if providing_args is None:
27             providing_args = []
28         self.providing_args = set(providing_args)
29     def connect(self, receiver, sender=None, weak=True, dispatch_uid=None): ...
98     def disconnect(self, receiver=None, sender=None, weak=True, dispatch_uid=None): ...
130     def send(self, sender, **named): ...
156     def send_robust(self, sender, **named): ...
193     def _live_receivers(self, senderkey):
194         """ ...
200         none_senderkey = _make_id(None)
201         receivers = []
202
203         for (receiverkey, r_senderkey), receiver in self.receivers:
204             if r_senderkey == none_senderkey or r_senderkey == senderkey:
205                 if isinstance(receiver, WEAKREF_TYPES):

```

Figure 5.2: An Example of Source Code for Context Collection

highly motivated to add this model to our study to check argument generation. Therefore, to generate a sequence of arguments, we have to collect a code context and a usage context of the variable of a source code.

5.3 Research Significance

Firstly, Our research was to check the overall performance of those three models and showed whether the models have enough information or context to recommend any arguments for a given method call; secondly, by identifying the categories where the models performed the worst to discover the development gap and propose a research field to improve these Pre-trained models. Thirdly, our approach also checked the performance of those three

models from the expression-type perspective and the precedence-based perspective.

5.4 Dataset

We collected Python projects from GitHub repositories based on specific criteria. A noble approach introduced a list of projects that are considered engineering projects [59].

To pursue our study, we collected our projects from GitHub (see-A.4 to understand the GitHub project download process) in a scrutinized way. We only considered projects with certain features, such as proper documentation, updated test cases, and a structured project management tool. To ensure these features, we collected our projects from the collection of Engineering projects [59]. Researchers in these projects collected and categorized the software repositories from GitHub using the following categories: belong to a community, have continuous integration, documentation, history, issues, star count, license, unit testing, and so on. The project list consists of 1,857,423 GitHub projects in different programming languages.

On top of their published lists of projects, we first filtered out those projects with Python as their primary language. As we worked with argument recommendations in the Python programming language, we filtered out the projects developed in Python. This resulted in 3,31,883 projects. Next, we checked whether the projects were forked projects or not and whether those projects were deleted or not. This step filtered out 37,339 projects. After this, we tried to select the active projects from the rest of the projects. To perform this, we collected only those projects with at least one commitment since last year. This step resulted in 14,437 projects. In addition to that, we considered a third metric: List of Contributors. We only selected those projects that had at least two contributors. This step gave us 10,717 projects in total. Finally, we have filtered out the projects by checking the number of commits, which is more than 50. This resulted in 9181 projects.

After sorting them by star count, we took the top 5000 projects with the maximum star count. The selected project list is shared with the community to support future research and study, which is in the following link.

5.5 Approach

The raw source code cannot be used to conduct our study. This source code needs to be extracted to collect features. Then, we must collect context from those features to generate a source code segment from the language models. After collecting the code segment, we have to collect the actual argument to compare with the real-time arguments. We followed the steps to conduct our study as follows:

- Data Extraction

- Context Collection
- Model Description and Use for Argument Generation.
- Similarity Comparison

5.5.1 Data Extraction and Preprocessing

We used the dataset mentioned in the section 5.4 for our study. We extracted the method call and its arguments for our experiment. Then, we used the method call and its context to generate an argument and pre-trained models for argument generation. For input generation of the pre-trained model, we followed the following steps:

- Method Call Extraction
- Global Variable Extraction
- Determine the scope of the method call

We must extract the method call and its relevant information from a source code for our analysis.

5.5.1.1 Method Call Extraction

We used the Python AST Parser [60] and followed the process described in section A.3 for method call extraction based on Python version 3.12. First, we parsed the entire file and generated an AST to access the node for collecting method calls. For a method call, we collected the **line of method invocation, name of the file, number of arguments, and the list of arguments**. For the method call “isinstance” at line 205 in the Figure 5.2, the argument list is [“receiver”, “WEAKREF_TYPES”], the number of arguments is two and the file name is “dispatcher.py”. We collected the method called node for further study as the node is an object that contains all the information encapsulated(see-A.2 for a pattern of a node) in a specific class.

5.5.1.2 Global Variable Extraction

We collected all the global variables from each file. These global variables have their scope everywhere in the file. It can be used inside any method definition.

Therefore, we extracted and added all the global variables to the method call context. As an example from Figure 5.2, for the method call “isinstance”, we added line no. 3 where the context is “WEAKREF_TYPES = (weakref.ReferenceType, saferef.BoundMethodWeakref)” as an additional context. to collect information about the global variable of a source file, we

parsed² and visited those nodes classified as Assignment nodes. This generated a list of all the variable assignments in a file. We generated a list of functions and class method ranges. In Figure 5.2, when we collected all the assignments, the assignment operations at lines 5, 31, 33, and 34 were extracted in the source context. As the scope of assignment operations at 31, 33, and 34 is only inside lines no. 24 to 34 for outside of the class method “__init_,” the variable cannot be used. Therefore, we removed those from the list of all variables, and the retained list of variables was treated as global variables.

5.5.1.3 Determine The Scope Of The Method Call

We collected those contexts for each method call only in the same scope. In Figure 5.2, for the method calls at line no 205, we had a method called “isinstance,” and it had two arguments, “receiver” and “WEAKREF_TYPES”. In the context of that method call, we collected the lines from 193 to 205. Therefore, for argument recommendation, we used tokens only in the scope of the method definition. It is mandatory to consider the scope of a variable as the usage of the variable outside the scope engenders software bugs [61]. Therefore, we considered the assignment operations in the same scope and the variable’s global scope (see 5.5.1.2 for variable in global scope).

5.5.2 Context Collection

We considered those tokens as the context of a method call, which was only in the scope of the definitions. This is known as a valid context. For the method call “isinstance” and recommending the first argument, the collected context will be all of the tokens, the lines from 193 to 205 until the token “receiver”. We collected the same tokens for the second argument but up to “WEAKREF_TYPES”. For both contexts, we have added line 3 as there is a variable in the file that can be used everywhere. We have removed all the comments from the context to remove the redundancy. These comments can be different from program to program and create ambiguity. Our study has three analyses, and they require two different contexts. We built the first dataset, which collects each argument, and the context is just before the argument. Table 5.1 showed how much context we collected for each argument for the given method call “isinstance” at Figure 5.2. The process of argument context generation was described in Figure 5.3.

We used the context for argument expression-wise analysis for our third study and generated the first argument. This first argument was added to the initial context to generate a new context for the following argument. Therefore, the output from the three models generated three different arguments for an initial position and then repeatedly added to the initial context to generate the next argument. This method is called the Argument

²<https://docs.python.org/3/library/ast.html>

Table 5.1: Collected Context for Expression-based Analysis and Overall Performance Analysis

Feature	Token Name	Expression Type	Method Call Line No	Context Remark	Line No	Collected Context
First Argument	receiver	Name	205	Variable Usage	3, 8-17	['WEAKREF_TYPES', 'weakref', 'ReferenceType', 'saferef', 'BoundMethodWeakref'], ['class', 'Signal', 'object', '_debugging', 'False']
				Local Context	205	['def', '_live_receivers', 'self', 'senderkey', 'none_senderkey', '_make_id', 'None', 'receivers', 'for', 'receiverkey', 'r_senderkey', 'receiver', 'in', 'self', 'receivers', 'if', 'r_senderkey', 'none_senderkey', 'or', 'r_senderkey', 'senderkey', 'if', 'isinstance']
Second Argument	WEAKREF_TYPES	Name	205	Variable Usage	3, 8-17	['WEAKREF_TYPES', 'weakref', 'ReferenceType', 'saferef', 'BoundMethodWeakref'], ['class', 'Signal', 'object', '_debugging', 'False']
				Local Context	205	['def', '_live_receivers', 'self', 'senderkey', 'none_senderkey', '_make_id', 'None', 'receivers', 'for', 'receiverkey', 'r_senderkey', 'receiver', 'in', 'self', 'receivers', 'if', 'r_senderkey', 'none_senderkey', 'or', 'r_senderkey', 'senderkey', 'if', 'isinstance',]

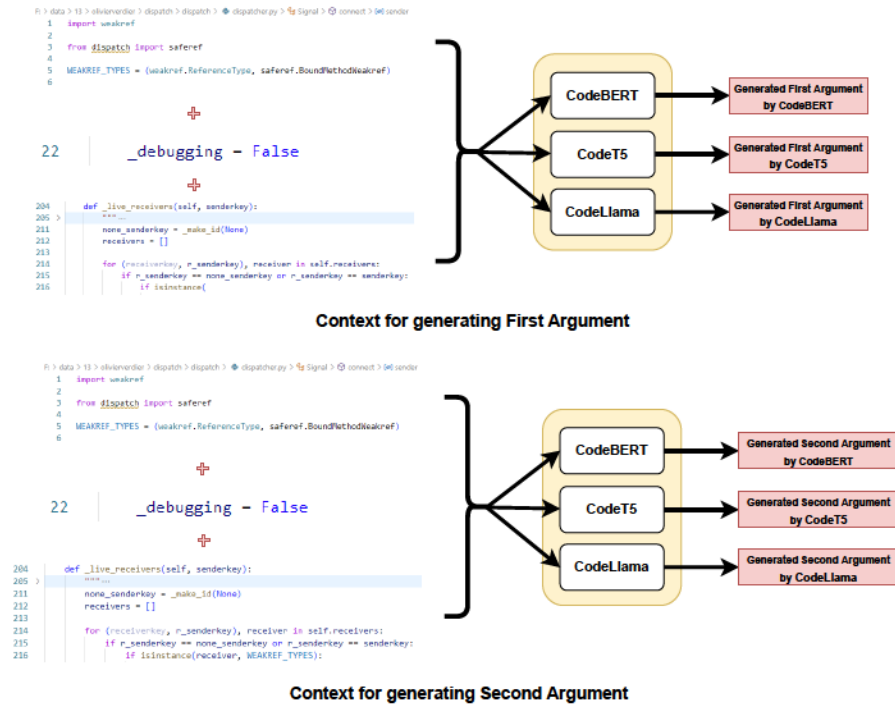


Figure 5.3: Input Context Generation for Argument Generation Based Study with CodeBert, Code Llama, CodeT5

Precedence approach which is shown in Figure 5.4.

5.5.3 Model Description And Using For Argument Generation

- CodeT5:** This encoder-decoder model was built on copious code data extracted from open-source GitHub repositories. This bimodal configuration takes a code segment and generates relationships between each word. For casual text generation tasks, it uses Text-Code Contrastive Learning (using a self-attention layer for a continuous bidirectional approach) [62], a text-code Matching (calculating the semantics information from similar code snippets), and an encoder-decoder for casual Large Language model operation. This model is trained with a dataset comprising millions of functions across multiple programming languages, including Python, Java, JavaScript, Go, PHP, and Ruby, and can catch subtle differences in different programming syntax and semantics. This model is used for Code Summarization, Code Generation, Code Translation, and Code Defect Detection.

We provided the context for generating an argument from a given context up to the argument we wanted to generate. First, we used the RobertaTokenizer tokenizer from the code5-base to tokenize the source code and collected the code context without comments. After collecting the contexts, we inserted these contexts as a query for the

Table 5.2: Expression Types With Their Examples In Python

Name of Expression	Frequencies	Example of Expression	Argument
Name	8658988	discovery_responder (msg , from_addr)	msg
Constant	7372512	resp_msg . decode ("UTF-8") == expected_response #	"UTF-8"
Attribute	2892357	sock = mock . create_autospec (socket . socket , instance = True)	socket . socket
Call	2338055	return st . tuples (st . just (ConvertChildrenToText ("StatusChange")) ,	st . just (ConvertChildrenToText ("StatusChange"))
BinOp	840814	assert (util . MetaInfo . from_meta_info (meta_info ["MetaInfo"] + "—extra") == expected	(meta_info ["MetaInfo"] + "—extra")
Subscript	751429	util . signal_strength_to_dbm (signal_strength ["SignalStrength"])	signal_strength ["SignalStrength"]
List	611626	subprocess . check_call (["sudo" , "—E" , "—u" , real_user , sys . executable , sys . argv [0] , "build"])	["sudo" , "—E" , "—u" , real_user , sys . executable , sys . argv [0] , "build"]
Tuple	311092	send_queue . put ((msg , addr))	(msg , addr)
Dict	285683	self . record_stats ({ "runs" : rendered_placeholders })	{ "runs" : rendered_placeholders }
JoinedStr	117070	except ProviderNot FoundException : self . stderr . write (f"* No OEmbed provider found for '{url}'!\n") except ProviderException as e :	f"* No OEmbed provider found for '{url}'!\n"
Starred	108792	def wrapper (* args , ** kwargs) : return function (* args , ** kwargs) return wrapper	* args
Compare	89360	x = cls (nu , h = hstart , N = int (np . pi / hstart)) . x lastk = np . where (f (x / np . max (K)) == 0) [0] if len (lastk) >1 :	f (x / np . max (K))
UnaryOp	88796	@ unittest . skipIf (not INTERNET , 'no internet') def test_ manpage_build_ without_warning (self) :	not INTERNET

Table 5.3: Expression Types with their examples in Python(contd.)

Name of Expression	Frequencies	Example of Expression	Argument
GeneratorExp	64175	continue elements . extend ((child , value) for value in value_or_list) if isinstance (name , ConvertChildrenToText) :	((child , value) for value in value_or_list)
ListComp	58402	ys = np . array ([lsq . update (x) [0] for x in xs])	[lsq . update (x) [0] for x in xs]
Lambda	49090	first_name = factory . faker . Faker (“first_name”) email = factory . Sequence (lambda n : “c%d@foo.com” % n)	(lambda n : ‘c%d@foo.com’ % n)
BoolOp	17173	stylefile = url base_path = os . path . abspath (self . base_path or os . curdir) if not os . path . isabs (stylefile) :	self . base_path or os . curdir
IfExp	12192) settings . load_profile (“ci” if os . getenv (“CI”) else “default”)	“ci” if os . getenv (“CI”) else “default”
Set	8240	set_attr_module({3, 4, 5, 6})	{3, 4, 5, 6}
DictComp	2592	compared_value({num: num**2 for num in numbers if num % 2 == 0})	{num: num**2 for num in numbers if num % 2 == 0}
SetComp	1388	get_data_user ({num**2 for num in numbers})	{num**2 for num in numbers}
Await	829	total_delay(await asyncio.sleep(2))	await asyncio.sleep(2)
Yield	223	user_IP_INFO(yield ip_address)	yield ip_address
YieldFrom	11	pr(yield from subgenerator())	yield from subgenerator()

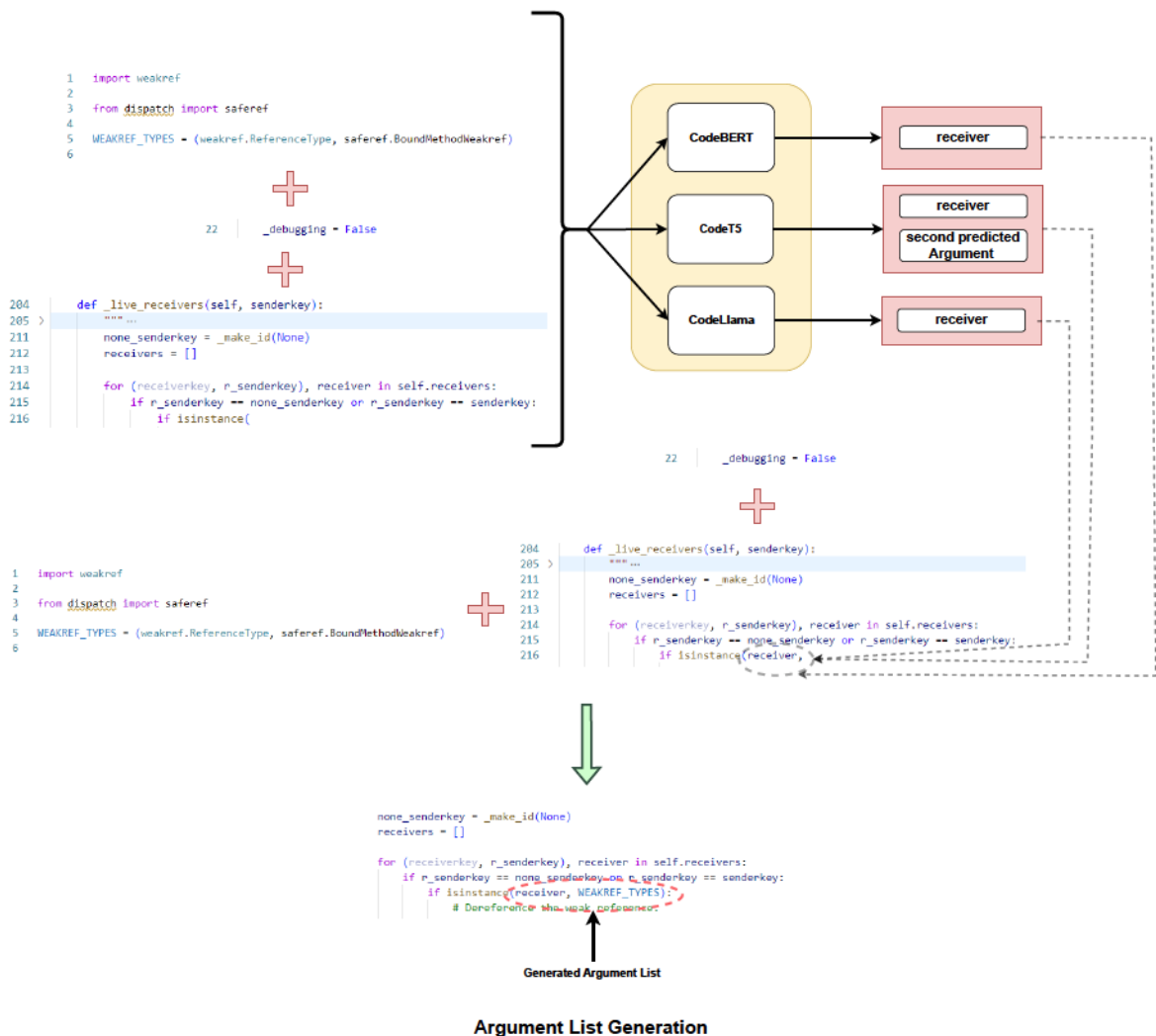


Figure 5.4: Input Context Generation for Argument Precedence Based Study with CodeBERT, Code Llama, CodeT5

CodeT5.

- CodeBERT:** CodeBERT is a transformer-based model with a bimodal architecture. The same as CodeT5, it can be used for code summarization, Code Generation, Code Translation, and Code Defect Detection. We used the same context extracted in the previous section for the input of CodeBERT [48]. As the CodeBERT is based on the contextual representation of each token and sequence, we used the context to generate the argument sequence. This model generates n-numbers of candidates, and any of them can be the argument. On top of BERT model [47] and the Roberta [63], for source code generation, CodeBERT performs with a multi-layer bidirectional

Transformer model (the same RoBERTa-base architecture with 125M parameters).

- **Code Llama:** The Code Llama models come in various sizes, with parameter counts ranging from 7 billion (7B) to 34 billion (34B) and even a larger variant with 70 billion (70B) parameters. These models are designed to handle a wide range of code-related tasks, including code generation, code completion, and understanding natural language instructions related to code. The architecture’s capacity to support large input contexts is particularly noteworthy, with the models being trained on sequences of 16,000 tokens and demonstrating improvements on inputs with up to 100,000 tokens. This capability is crucial for processing complex codebases and providing relevant code generations or completions based on extensive context.

We used the pre-trained and fine-tuned model from the hugging face library³. We used the pre-trained model from three pre-trained models, the base version with 13B parameters model [64]. We used the same context from the previous section 5.5.2. This model does code infilling, which is used for code missing part generation from the surrounding context, and long context fine-tuning, which generates the tokens from a long sequence. Therefore, our collected context is used to create the sequence effectively.

5.5.3.1 Input Generation for Models for Evaluation:

Earlier in our approach, we collected the code context by adding the usage context available and the block information as input for our model. We used three different approaches to generate the arguments for three different models.

For the first model, we generated the input token-wise. From the Figure 5.2, to generate the first argument of “isinstance”, we collected the tokens before the argument “receiver” with all the global variables (as example-WEAKREF_TYPES). For the generation of the second argument, we used the tokens before “WEAKREF_TYPES” with all the global variables (for example- “WEAKREF_TYPES”). Therefore, these three models get context from one token after another token. Figure 5.4 shows how we collected and passed context for three different models.

Our second research question was to find the model’s performance for individual expression types. Our analysis showed that we had 26 commonly used argument expressions. For your studies, we collected 200 unique examples from each expression type. Therefore, we had 2600 unique calls with their expressions. We used the same approach described in the previous section, though we sampled them for their expression types. Some expressions, such as Name, constant, and Call, are important and relatively straightforward to generate. On the other hand, some of the expressions are unpredictable, and we cannot generate

³<https://huggingface.co/codellama/CodeLlama-13b-hf>

them from a given code context. In the example in Table 5.2 and Table 5.3, we showed different expression types with their examples. Therefore, we considered the 24 categories of frequently used arguments from the 5000 Python projects. Therefore, we considered 200 examples from each category and generated a dataset for argument generation. We used the same context and generated the arguments.

For our third analysis, we checked the performance of these three models for a precedence-based result. We aim to check that these three models can generate the complete list of arguments for a given method call. For this evaluation, we used a recursive generation of context and checked the performance of these models. We took 5000 examples and used them to generate one after another argument.

5.6 Evaluation Procedure

Our evaluation procedure is based on three performance analyses of the three-language models.

- **RQ1:** Can we propose a taxonomy based on the expression type?
- **RQ2:** Can Large language Models perfectly recommend arguments from a given context for Python Programming? What is the performance of pre-trained large language models for generating any argument for any method calls in Python?—Comparison of CodeT5, CodeBERT, and Code Llama models.
- **RQ3:** What is the accuracy of the pre-trained model for generating arguments individually for each expression type?
- **RQ4:** How accurate are these language models' performance in generating the list of arguments for that method calls?

5.6.1 Evaluation Metrics

We used string similarity matrices to check the exact match of the actual and generated argument lists. If the similarity is greater than 0.5, we consider it a correct match. To check the match for expression type base analysis, we considered manual analysis, where we followed three rules to check the match of generated arguments. Firstly, we checked to see if it matches the actual list of arguments. Secondly, we checked the generated argument list with the global argument list. Thirdly, we checked the argument list with the local variable list.

5.6.2 Performance Comparison of CodeT5, CodeBERT, Code Llama Models

Motivation: CodeT5, CodeBERT, and Code Llama can generate tokens from a given code sequence. We aim to use our collected context and check whether these models can generate tokens from them. If we get tokens from each model, we compare the tokens after identifying the argument list. We generated arguments for a method call regardless of their position and expression types. In any project, the usage of the expression is indefinite. Therefore, our first aim was to investigate the performance of overall method calls and their arguments. For a given context, CodeBERT generated a single argument; codeT5+ generated two different argument recommendations. Code Llama generated a sequence of tokens.

Study Procedure: We used the tokens passed to the three models and collected the generated tokens for each model. We used the string similarity to check the match between the generated tokens and actual arguments. For CodeBERT and codeT5+, we did not process the generated tokens as they were the required tokens. On the other hand, we used an AST parser to get the argument segment from the generated token Code Llama.

Similarity Comparison: Investigating whether the arguments match the original argument is time-consuming and complex. To overcome this complexity, we used Levenshtein Distance and cosine similarity, Jensen-Shannon Divergence (JSD), and Jaccard similarity analyzers to compare the similarities between ground truth and generated arguments. For the Levenshtein distance, we considered two of the words from the context as input and calculated the total number of single characters to change one word to another. This metric was especially valuable in scenarios requiring the detection and correction of spelling variations, like text editors, search engines, and data entry validation systems. The Jensen-Shannon Divergence (JSD) is a technique for quantifying the likeness between two probability distributions. It's a smoothed and symmetrized adaptation of the Kullback-Leibler divergence (KL divergence), which gauges the difference between one probability distribution and an anticipated one. JSD is computed as the average KL divergences between the two distributions and their mean. In the equation 5.2, where $M = (1/2)(P+Q)$.

$$JSD(PQ) = \frac{1}{2} * [KL(PM) + KL(QM)] \quad (5.2)$$

The Jaccard similarity, or Jaccard index, is a statistical measure used to assess the similarity between two sets. It is calculated as the size of the intersection of the sets divided by the size of the union of the sets. Mathematically, given two sets of `org_arg` and `tokens`, the Jaccard similarity $JD(\text{org_arg}, \text{tokens})$ is defined in the equation 5.3. Here, $|\text{org_arg} \cap \text{tokens}|$ implied that the number of elements common to both sets (the intersection

) and $|org_arg \cup tokens|$ represented the number of distinct elements in both sets (the union).

$$JD(org_arg, tokens) = |org_arg \cap tokens| / |org_arg \cup tokens| \quad (5.3)$$

We set a threshold value of 0.5 and considered it a correct prediction if it is more than 0.5. To check the similarity, this followed three steps (see 5.6.1). Cosine similarity was a method for measuring the similarity between two non-zero vectors in an inner product space. It computes the cosine of the angle between the vectors, yielding a value between -1 and 1. A cosine similarity of 1 indicates that the vectors are identical, 0 signifies orthogonality (perpendicularity), and -1 suggests that the vectors are in opposite directions (see equation 5.4). In the equation 5.4, $(org_arg \cdot tokens)$ is the dot product of the two vectors, the L2 norm (Euclidean length) of a vector org_arg is typically denoted as $\|org_arg\|$, and similarly, the L2 norm of a vector $tokens$ is denoted as $\|tokens\|$. We calculated the similarity values by considering the equations and used a threshold value greater than 0.5 to determine a correct prediction.

$$cosine_similarity(org_arg, tokens) = (org_arg \cdot tokens) / (\|org_arg\| * \|tokens\|) \quad (5.4)$$

Table 5.4: Over All Model Performance Model Performance

Similarity Evaluation Matrices	Model Performance		
	CodeT5	CodeBERT	Code Llama
Levenshtein Distance [65]	45	46	59
Cosine similarity [66]	41	43	58
Jensen-Shannon Divergence (JSD) [67]	53	58	66
Jaccard similarity [68]	48	52	57

Result: All of the models performed poorly. The model CodeLlama performed more than other models. The result was checked manually after data generation, and if the generated token was found in the context and if it was the name of a variable, we put that in a match. Table 5.4 shows the result of each model to their similarity matrices.

5.6.3 Expression-wise Performance Comparison of CodeT5, CodeBERT, Code Llama Models

Motivation: To investigate the performance of each expression type, we calculated the accuracy for each type by manually validating 4800 examples. Our goal was to provide an idea to notify the developers why and which type of expression type is hard to generate or

recommend.

Study Procedure: In the section 5.5.3.1, the collected results were split into expression-wise. Then, we check which expressions can be detected by the Large language model Table

Table 5.5: Expression wise- Model Performance

Expression Type	Model Performance		
	CodeBERT	Code Llama	CodeT5
Name	62	89	70
Starred	64	88	48
Subscript	42	87	39
Compare	51	83	48
Attribute	53	83	53
Call	42	80	56
Tuple	52	77	40
UnaryOp	46	46	26
List	25	45	22
Dict	25	44	25
Constant	15	43	22
BinOp	16	19	8
GeneratorExp	5	5	5
ListComp	0	0	0
DictComp	0	0	0
Await	0	0	0
BoolOp	0	0	0
Set	0	0	0
SetComp	0	0	0
IfExp	0	0	0
JoinedStr	0	0	0
Yield	0	0	0
Lambda	0	0	0

5.5 showed the performance of the Large Language model expression-wise. Our three models performed well for Name, constant, and attribute types. However, the result is significantly lower for complex expressions such as Lambda, BoolOp, IfExp, Set, DictComp, Yield, SetComp, Await, Yield, NamedExpr, and YieldFrom.

Result: The Table 5.5 showed the performance of each model for their expression types. The table showed that for the first 13 expression type, the result was promisingly good, whereas we found ‘BinOp’, ‘GeneratorExp’, ‘ListComp’, ‘NamedExpr’, ‘DictComp’, ‘Await’, ‘BoolOp’, ‘Set’, ‘SetComp’, ‘IfExp’, ‘JoinedStr’, ‘Yield’, ‘Lambda’ showed less than 30% accuracy. We tried our best to match the content of the global variable and generate the sequence.

Reason of Low accuracy:

- “GeneratorExp”, “ListComp”, “SetComp”, and “DictComp” implied that a list comprehension returns a list, Set, and dictionary. The main advantage of a generator expression is that it will not save the list in the memory. Therefore, this is a user-defined and arbitrary list generator that depends on the usage context of a source code. We manually investigated ten different generator expressions, “Dict Comparison” and “List Comparison” which do not follow any pattern.
- The “await” keyword is used to pause the execution, and usage of this keyword entirely depends on the user itself. Therefore, the usage of awaits as an argument is also unpredictable. Similarly, the “yield” statement is used inside a function to return a generator, which can be iterated to produce a sequence of values. Therefore, this can be replaced by any loop statement. Thus, this cannot be understood by the Large language model as those models cannot catch the context of a source code.
- “Lambda” Expressions are inline precise forms of complex expressions. When a large language model generates a sequence from a context, it is unpredictable that this complex expression must be kept short. Therefore, our existing Large language model failed to generate these tokens. Similarly, “BoolOp” can not be expanded to multiple lines and can be used arbitrarily.

5.6.4 Argument Precedence based Performance Comparison of CodeT5, CodeBERT, Code Llama Models

Motivation: There can be multiple arguments in a method call. For a correct method call, it is mandatory to pass the arguments correctly. Therefore, our third analysis checks the performance of the models for precedence-based prediction analysis. We aimed to investigate the performance of three models to detect all method arguments correctly. Our study also checked whether different generated arguments changed the following argument. In the given example, the first argument is “receiver”. If the model detects the first argument, it will typically suggest the next one based on the context and patterns it learned. However, if the first argument is incorrect, the model might still attempt to suggest the next argument, but it might not be accurate due to the incorrect starting point.

Study procedure: We used a model for each example n number of times. Here, n is the number of arguments in a method call. It is a recurrent combination of large language models that generate and add an argument with the previous context. First, we collected the context for the first argument by the above-mentioned process. Then, we fit that context, which generated the first argument. After generating the first argument, we processed the generated text to remove unnecessary tokens and added the predicted actual argument

Table 5.6: Model Performance-Precedence-wise Result

Similarity Evaluation Matrices	Model Performance		
	CodeT5	CodeBERT	Code Llama
Levenshtein Distance	43	41	53
Cosine similarity	40	41	52
Jensen-Shannon Divergence (JSD)	51	52	61
Jaccard similarity	43	51	55

with the first context. This new context was used as an input to generate the following argument. This approach generated a sequence of arguments with the same number of method arguments in the actual method call. Our study then used the same similarity-based approach with a manual study to check the performance.

5.7 Result

Table 5.6 shows the performance of those three models. We found that Code Llama produced the most correct prediction among the three models. Compared to the table 5.4, the performance of the precedence-based model is less due to the dependency of the next argument on the before argument. If the before argument is incorrect sequentially, the next argument may lead to a wrong prediction. Besides, this sequence generation showed among the 4800 examples, 2% of the results were in the wrong sequence of argument, which leads to swapping argument-related bugs. The reason behind low accuracy is the combination of different expressions, according to the combination of different expression types at 4.6.5, which argument will be passed when and where is unpredictable. Therefore, Table 5.6 showed the performance of the three models is very poor, and a combination of those expression types will produce poor results. Based on the result, we built a classified taxonomy found at 5.5. We categorized the examples based on the pattern of expression types.

5.8 Taxonomy of Argument Types For Future Research

We categorized all of the expressions into five major categories. These categories are-

- Variables
- Operations
- Sub-Functions
- Key-Value Pairs

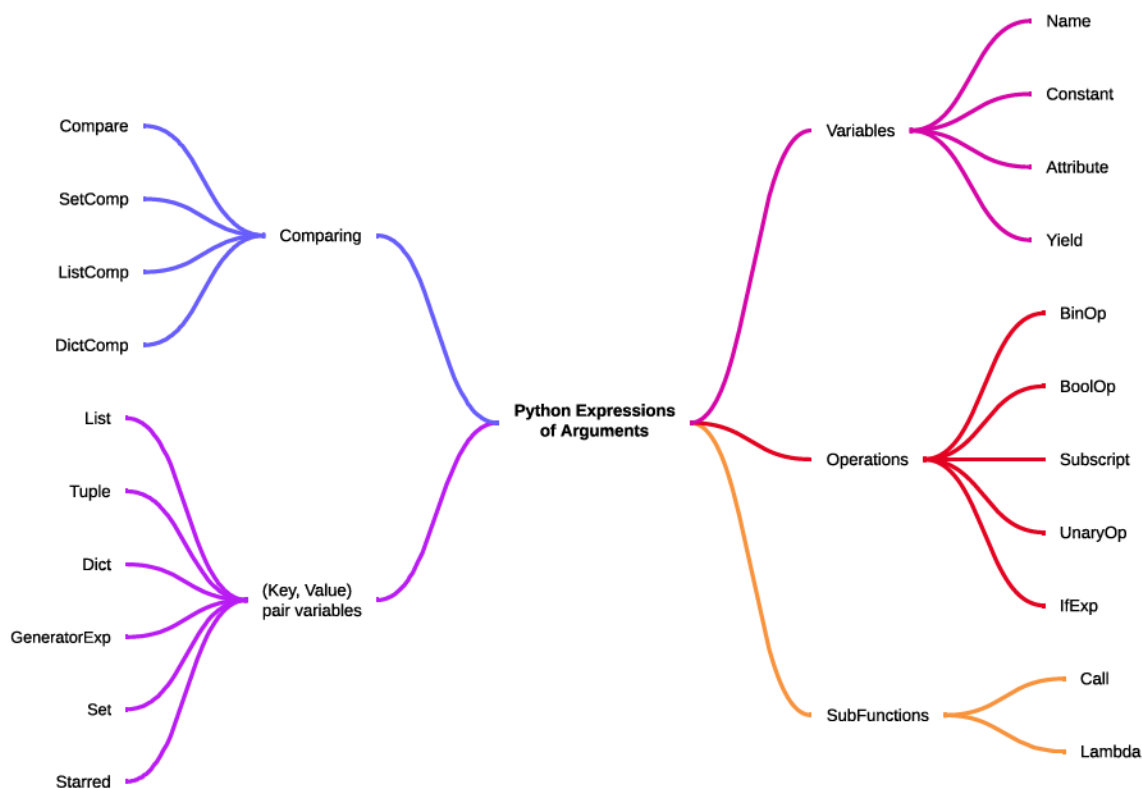


Figure 5.5: Categorization of Expression Type

- Comparing

According to Table 5.5, we found the categories Variables and operations are more predictable and can be generated from a context. The large language models generate call expressions as the calls are associated with code context. The models are trained based on call-to-call dependencies. All of the models performed very poorly for compare operations. From Table 5.3, we found examples of SetComp, ListComp, and DictComp where the programmers are unsure of the logic they should use. Future models should be trained with that context, mined from the documentation and source code context to solve this issue. Therefore, future research can be carried out to predict the argument lambda statements, SetComp, ListComp, and DictComp.

5.9 Conclusion

Large language models are trained with their context information. This context is quite unknown to the user. Therefore, when we use the models, there is a high chance of passing

the wrong context sequence to generate an argument. Even if we passed the context correctly, how and what the parameters used to balance the weight of LLM are unknown to the developer. For better performance, the model expects a similar context. It is tough to predict when the arguments are highly complex and user-defined. On the other hand, the frequency of complex argument expressions reduces the performance of the models. However, it performed better for call, name, and attributes as these methods' arguments are associated with context and follow a common usage pattern. Our study showed which patterns are intricate to determine, and this study can be extended by proposing extra feature collection for those expressions so that they can be identified quickly. Our future work will find the pattern and collect extended context as a feature for expressions with low accuracy. Another aspect of our study is effectively using a large language model when variables or tokens are commonly used. Fine-tuning these large language models is not quicker than other deep learning techniques.

Chapter 6

Conclusion

This chapter discusses the summary of the studies presented in the previous chapters along with future research directions.

6.1 Summary

Exploring name-based bug detection shows we can treat a source code as a regular textual block. When we consider only the words of a source code, the performance drops as the deep learning model cannot discriminate the nuance among the tokens. Therefore, extended features or context must be introduced to identify the code tokens uniquely. Our thesis first proved that an approach is not always effective for all the programming languages for variation of structural information and grammatical patterns. Thus, this approach was improved by adding and verifying extended features of the variable (usage context). This introduced the importance of programming context and showed that better name-based information could be hidden through a token. Though our approach did not consider the method definition information, it followed the name-based approach for detecting the swapping argument-related bugs by considering the name-based usage context information from a source code. Besides, a token may have multiple uses, and to discriminate the name information, we added the structural information to the tokens to identify those tokens uniquely. We determined the difference between the variables for the same variable name from the usage context. Finally, our thesis showed the importance of different sources of information related to the features of the method call and investigated the performance of various models based on source information. This thesis also showed the performance of models based on the expression type of arguments and brought up that our model performs for any expression type. Our thesis also conducted an empirical study to show which expression types are hard to detect.

At first, our thesis solved and proposed a technique to detect the swapping argument-

related bugs in Python and reduce the complexity of dynamically typed programming language (type dependency at run-time). Secondly, this thesis checks the performance of the existing large language models in generating arguments. We implemented three different techniques to test the performance of large language models from three aspects and also showed the reason for the poor performance based on expression types. From this analysis, we posit that all the expression types do not follow a specific pattern.

6.2 Future Work

This thesis not only proposed a technique for swapping argument-related bug detection in Python but also opened some of the research fields for dynamically typed programming languages. From our studies, we found several research gaps in dynamic programming languages. Exciting opportunities for further advancement and refinement in this field of study offer promising prospects for future research. Based on our search, the following enhancements can be done:

- **Exploring Other Programming Languages:** The same as Python, this approach can be used in another programming language- R, Ruby, Perl. As the structure of Python is most likely to be Ruby and R, our next analysis is to find the performance of those programming languages.
- **Dataset Scalability:** Our Data was trained on 2 million Python examples. Shortly, we will propose our dataset, where we will check the performance of the AUCMET Model for 10 million Python examples. If a larger dataset provides the same accuracy, it can be derived that a large-scale study can be done with extra structural information. Failing to reproduce the accuracy on a large scale may lead us to explore new features and detect the expression where the process performs badly.
- **Proposing Usage Pattern-Based Study for Dynamic Programming Language:**Our thesis is the first approach to consider the variable and argument usage pattern to detect the bug, which can enhance usage pattern-based anomaly and bug detection in any programming language.
- **Comparing with LLM Models:** Comparing the AUCMET Model with trending large language models is not performed here; therefore, comparing the AUCMET Model with LLM models will allow the researcher to enhance LLM in the near future.
- **Vocabulary Size Scalability:** The more training Examples, the more vocabulary and variations will be found. If the large dataset generates any issue in performance, one of the reasons can be the large scale of vocabulary, which can be handled by

considering a threshold value of a highly frequent list of vocabulary. Therefore, the performance of the AUCMET Model can be tuned to the vocabulary size.

- **Tool Paper (ArgPatt):** A warning-generating tool for Python can be built as our trained model performed very well for unseen datasets. Therefore, it can be a lightweight enhancement of any Python IDE, generating a warning when an argument is called in the wrong sequence.
- **Argument Recommendation With LLM:** Our Second work performed an empirical study and showed that the performance is poor for some expression types. Therefore, it can be a part of a new analysis of how to enhance the code sequence generating for the arbitrary patterns in Python.

Appendix A

Installation of Modules and Environment Setup

A.1 Installation and Update Ubuntu

Update Ubuntu Using the Command Line

For server environments or those who prefer using the terminal, we used to update Ubuntu by following these steps:

1. Open your terminal.
2. First, update the package list to inform your system about the latest versions of packages and their dependencies by running:
`sudo apt update`
3. Then, upgrade all the installed packages to their latest available versions with:
`sudo apt upgrade`

A.2 Required Modules

We used the following modules for our research-

- Abstract Syntax Trees AST ¹

Installation Command: `pip3 install AST`

¹<https://docs.python.org/3/library/ast.html#module-ast>

- Astor – AST observe/rewrite ²
Installation Command: pip3 install Astor
- Git clone ³
Installation Command: sudo apt update
sudo apt install git
- seaborn ⁴
Installation Command: pip install seaborn
- Matplotlib ⁵
Installation Command: python -m pip install -U matplotlib
- tokenize ⁶
Installation Command: This is a built-in Python Module which is integrated with Python>=3.8
- tqdm ⁷
Installation Command: pip install tqdm
- pandas ⁸
Installation Command: pip install pandas
- numpy ⁹
Installation Command: pip install numpy
- scikit-learn ¹⁰
Installation Command: pip install -U scikit-learn
- keras ¹¹
Installation Command: pip install --upgrade keras
- gensim ¹²
Installation Command: pip install gensim
pip install --upgrade gensim

²<https://astor.readthedocs.io/en/latest/>

³<https://www.digitalocean.com/community/tutorials/how-to-install-git-on-ubuntu>

⁴<https://seaborn.pydata.org/>

⁵<https://matplotlib.org/>

⁶<https://docs.python.org/3/library/tokenize.html>

⁷<https://pypi.org/project/tqdm/>

⁸<https://pandas.pydata.org/>

⁹<https://numpy.org/>

¹⁰<https://scikit-learn.org/stable/index.html>

¹¹https://keras.io/api/layers/regularization_layers/dropout/

¹²<https://radimrehurek.com/gensim/models/word2vec.html>

- tensorflow¹³
Installation Command: `pip install tensorflow`
- CodeT5-base¹⁴
Command for Downloading Model :

```
tokenizer = RobertaTokenizer.from_pretrained('Salesforce/codet5-base-multi-sum')
model = T5ForConditionalGeneration.from_pretrained('Salesforce/codet5-base-multi-sum')
```
- CodeBERT¹⁵
Command for Downloading Model :

```
model = RobertaForMaskedLM.from_pretrained('microsoft/codebert-base-mlm')
tokenizer = RobertaTokenizer.from_pretrained('microsoft/codebert-base-mlm')
```
- codellama¹⁶
Command for Downloading Model :

```
model = AutoModelForCausalLM.from_pretrained("codellama/CodeLlama-7b-hf")
model = accelerator.prepare("codellama/CodeLlama-7b-hf")
```
- Accelerate¹⁷
Installation Command: `pip3 install accelerate`

A.3 Parsing With Python AST

Every language has some predefined rules. These rules are mandatory to use a language perfectly. Let's think about languages like English, Bengali, or French. We need to generate verbal speech using a rule known as grammar. Similarly, in the programming field, the programmers follow specific rules so that both the programmer and compiler understand the written script. Now, how will the grammar analyze the code script? Let's think about a method call in Python.

```
greet ("John")
```

But how does the compiler decode this line “greet(“John”)”? The compiler will read the

¹³<https://www.tensorflow.org/install/pip>

¹⁴<https://huggingface.co/Salesforce/codet5-base-multi-sum>

¹⁵<https://huggingface.co/microsoft/codebert-base-mlm>

¹⁶<https://huggingface.co/blog/codellama>

¹⁷<https://huggingface.co/docs/accelerate/en/index>

word “greet” as a name, and then after that name, if it finds parentheses, it will expect a list of arguments and a closing parenthesis. The reading pattern of a compiler will be `func_name (OPEN_PAREN arglist? CLOSE_PAREN)`

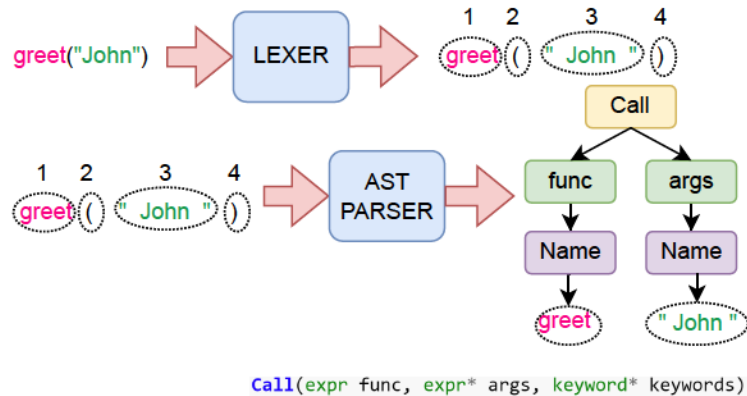


Figure A.1: Parsing With Python AST

Therefore, a typical parser consists of two components: a lexer and a parser. The functionality of a lexer is to analyze the code script and tokenize that with respect to the keywords and token. Then, it goes through the token sequence with respect to its grammar. In the parser, the Syntactic and Semantic analyzer determine the meaning and the pattern. During this parsing phase, the input’s syntactic structure is examined using a data structure known as a parse tree or derivation tree. A syntax analyzer utilizes tokens to build a parse tree, merging the predetermined grammar of the programming language with the tokens from the input string. If there are any syntactic errors, the syntax analyzer will flag them and report them as such. Semantic analysis involves validating the parse tree against a symbol table to ensure semantic coherence. This step is also referred to as context-sensitive analysis. It encompasses tasks like checking data types, verifying labels, and validating flow control.

Extracting Features From Scripts: We treat the script as natural language. Therefore, we generated an Abstract Syntax tree to traverse the individual tokens and collect data. We used the Python “ast” library to collect the required information from the code. AST module has a method `parse()`, which will compile a file and parse after checking lexical, syntactic, and semantical errors. The parsing provides a class-wise token classification. As Example: We want to generate an AST for a Python script. We will collect the file path to open the file.

The `patch.py` file has the content as follows.

In the above example, the file is opened in the first line, and then the file is read by the `read()` method, which takes the whole file as a text file. But we need to go through the code as a collection of classes. To do that, we have used the `ast.parse()`, which goes through the

```

parsing.py
Dr > Type_detection > testing > project_1 > ajul_pyradox > scripts > hoi4 > parsing.py > ...
1 import ast
2 with open(r"F:\Data_15-k\data\data\00\wikihouse\patch.py", encoding='utf-8') as f:
3     parentDict={}
4     code=f.read()
5     node=ast.parse(code)

patch.py 4
F:\Data_15-k\data\data > data > 00 > wikihouse > patch.py
1
2 if not third_party_dir in sys.path:
3     sys.path.insert(1, third_party_dir)
4

In 2 1 import astor
2 print(astor.dump_tree(node))

Module(
  body=[
    If(
      test=UnaryOp(op=Not,
        operand=Compare(left=Name(id='third_party_dir'),
          ops=[In],
          comparators=[Attribute(value=Name(id='sys'), attr='path')])),
      body=[
        Expr(
          value=Call(
            func=Attribute(value=Attribute(value=Name(id='sys'), attr='path'), attr='insert'),
            args=[Constant(value=1, kind=None), Name(id='third_party_dir')],
            keywords=[]),
          otherwise=[]),
        type_ignores=[]
      ]
    )
  ]
)

```

Figure A.2: Showing Instances of a Node

text and collects the information by first tokenizing the code by the lexical analyzer and later collecting the information of the code by building an abstract syntax tree. Typically, Python has 19 classes. All the Python classes are part of the module as a form of parsed tree, which refers to the Python source code or the current file. If we print the “node,” we will see an AST class with the information of the certain file but as a parsed tree. Let’s go through what is inside the `ast.Module`. No doubt! It will carry all the information in the file as a parsed tree. To print the information in the `ast.Module`. We take help from another library known as “astor” and the “`astor.dump_tree()`”

```
import astor
print(astor.dump_tree(node))
```

This will give a visual output of the AST tree as follows. Let’s match the AST with the code snippets. In the code snippet, there is an “if” statement in the body of the module, and when the code is parsed, the AST is generated in such a sequence that which node is inside which node can be depicted. Thus, the snippets are as follows. A call node \subset an Expression node \subset the body of If Expression \subset Body of module \subset *Module*.

Therefore, if we want to access any node, we must visit the nodes by their certain class.

Traversing or visiting a Node: In the AST library, we have a class known as class `ast.NodeVisitor`. It walks the generated AST and invokes a predefined visitor function for nodes. For example, if we want to visit a node with a method call, we must call the `visit_Call(self, node)`. Thus, we can add different visitors according to our needs and jump to certain types of nodes to gather information.

Random visit: `generic_visit(node)`

This certain visit walks all the nodes, especially the children nodes, and reaches the leaf node. This visitor calls `visit()` on all children of the node. Traversing through the node will be implemented as follows.

- Define a class and extend the class “class_name” by the `ast.NodeVisitor` (will allow any modification of nodes).
- Define the `visit()` method in the class (class_name). As example

```
1 def visit_Call(self, node):
2     # method body functionalities
3     return node
```

Figure A.3: Visiting A Parsed Node

- Define a variable to collect the required data from each iteration and save them. As example

```
D: > Type_detection > testing > project_1 > ajul__pyradox > scripts > hoi4 > parsing.py > ...
1 import ast
2 call_list_collector=[] #a list to collect all the visited call nodes.
3
4 class Class_as_parent(ast.NodeVisitor):
5     def visit_Call(self, node):
6         #method body functionalities
7         call_list_collector.append(node)
8         return node
```

Figure A.4: Visiting A Parsed Node by Class method

- By getting all the required nodes from a file, we will follow the abstract grammar to learn the possible nodes found with the parent node and collect the data. As example: If we use `astor.dump_tree`, we can see the node data inside the curtain call as follows:
- Now we know from the abstract grammar we have a function and an args (the argument list). To split them, we have our node analyzer function, which returns the information based on the requirement. Therefore, if we run the `call_node_analyzer`


```

Call_node
<ast.Call object at 0x0000023392DF4730>
<ast.Call object at 0x0000023392DF45E0>
<ast.Call object at 0x0000023392DF43D0>
<ast.Call object at 0x0000023392DF42B0>
<ast.Call object at 0x0000023392DF4040>
<ast.Call object at 0x0000023392DF3E80>
<ast.Call object at 0x0000023392DF3A90>
<ast.Call object at 0x0000023392DF3800>

print(astor.dump_tree(df_call_merge['Call_node'][0]))

Call(func=Name(id='require_setting'),
      args=[Constant(value='static_url_prefix', kind=None)],
      keywords=[keyword(arg='default', value=Constant(value='/static/', kind='u'))])

print(df_call_merge['Call_node'][0])
print("line_number-->",df_call_merge['Call_node'][0].lineno)

<ast.Call object at 0x0000023392DF4730>
line_number--> 24

```

Figure A.5: Result of Parser

function, it will return the name of the function as “require_setting” and the argument value as [“static_url_prefix”].

- We can get the location or line number of the method call by using `node_object.lineno` from the `ast.Call` object.

A.4 Downloading Projects from GitHub

We used the `git` library and `git clone` command to download the projects from GitHub. Figure A.6 shows the source code of how we can download projects to the local machine.

A.5 Reproducing the study Exploring Name-based Bug Detection in Python

We have uploaded the dataset and the source code to the folder Google Drive “Exploring Name-based Bug Detection in Python”¹⁸. The execution workflow is given below-

¹⁸https://drive.google.com/drive/folders/1J7YMyvUyoebd7PFUQny6I6cLAI1_pzPy?usp=drive_link

```

1 import pandas as pd
2 import subprocess
3
4 from tqdm import tqdm
5
6 df=pd.read_json('json_file_of_all_the_project_git_link',orient='index')
7 project_list=df['projects'].to_list()
8 import os
9
10 save_path=[local_machine_save_path/'
11
12 for path in tqdm(project_list):
13     try:
14         username = 'user_name'
15         password = 'github_password'
16         print("path--->",path.replace("/","__"))
17         os.makedirs(save_path+"/"+path.replace("/","__"))
18         dxp=save_path+"/"+path.replace("/","__")
19
20         clone_command = f'git clone https://{username}:{password}@github.com/{path}.git {dxp}'
21
22         result = subprocess.run(clone_command, shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
23     except Exception:
24         pass
25

```

Callouts in the image point to the following URLs:

- `https://github.com/pandas-dev/pandas.git`
- `https://github.com/numpy/numpy.git`

Figure A.6: Process of Git Repository Clone

A.6 Reproducing the study Empirical Study of Argument Recommendation by LLM in Python

We have uploaded the dataset and the source code to the folder Google Drive “Empirical Study of Argument Recommendation by LLM in Python”¹⁹.

¹⁹https://drive.google.com/drive/folders/1lbSjKUKa7titY3GP9j7Z1kD10oXt0-hM?usp=drive_link

Bibliography

- [1] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, “Exploring the influence of identifier names on code quality: An empirical study,” in *Proceedings of the 14th European Conference on Software Maintenance and Reengineering*, 2010, pp. 156–165.
- [2] M. Pradel and K. Sen, “Deepbugs: A learning approach to name-based bug detection,” *ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–25, 2018.
- [3] R. Gupta, S. Pal, A. Kanade, and S. K. Shevade, “Deepfix: Fixing common C language errors by deep learning,” in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, 2017, pp. 1345–1351.
- [4] D. J. Lawrie, C. Morrell, H. Feild, and D. W. Binkley, “What’s in a name? A study of identifiers,” in *Proceedings of the 14th International Conference on Program Comprehension (ICPC)*, 2006, pp. 3–12.
- [5] C. D. Newman, R. S. Alsuhaibani, M. L. Collard, and J. I. Maletic, “Lexical categories for source code identifiers,” in *Proceedings of the IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER*, 2017, pp. 228–239.
- [6] J. Patra and M. Pradel, “Nalin: learning from runtime behavior to find name-value inconsistencies in jupyter notebooks,” in *Proceedings of the 44th IEEE/ACM 44th International Conference on Software Engineering*, 2022, pp. 1469–1481.
- [7] S. Kim, J. Choi, M. E. Ahmed, S. Nepal, and H. Kim, “Vuldebert: A vulnerability detection system using BERT,” in *Proceedings of the IEEE International Symposium on Software Reliability Engineering Workshops*, 2022, pp. 69–74.
- [8] N. Ziems and S. Wu, “Security vulnerability detection using deep learning natural language processing,” in *Proceedings of the 2021 IEEE Conference on Computer Communications Workshops, INFOCOM Workshops*, 2021, pp. 1–6.
- [9] T. Index, “Tiobe,” 2021. [Online]. Available: <https://www.tiobe.com/tiobe-index/>

- [10] M. Pradel and T. R. Gross, “Name-based analysis of equally typed method arguments,” *IEEE Trans. Software Eng.*, pp. 1127–1143, 2013.
- [11] A. Rice, E. Aftandilian, C. Jaspan, E. Johnston, M. Pradel, and Y. Arroyo-Paredes, “Detecting argument selection defects,” *Proc. ACM Program. Lang.*, pp. 104:1–104:22, 2017.
- [12] J. Gao, E. Xun, M. Zhou, C. Huang, J. Nie, and J. Zhang, “Improving query translation for cross-language information retrieval using statistical models,” in *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2001, pp. 96–104.
- [13] B. Berabi, J. He, V. Raychev, and M. Vechev, “Tfix: Learning to fix coding errors with a text-to-text transformer,” in *Proceedings of the International Conference on Machine Learning*, 2021, pp. 780–791.
- [14] M. Allamanis, H. Jackson-Flux, and M. Brockschmidt, “Self-supervised bug detection and repair,” in *Proceedings of the Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems*, 2021, pp. 27 865–27 876.
- [15] M. Acharya, T. Xie, J. Pei, and J. Xu, “Mining API patterns as partial orders from source code: from usage scenarios to specifications,” in *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM*, 2007, pp. 25–34.
- [16] Y. Wainakh, M. Rauf, and M. Pradel, “Idbench: Evaluating semantic representations of identifier names in source code,” in *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering, ICSE*, 2021, pp. 562–573.
- [17] H. Liu, Q. Liu, C. Staicu, M. Pradel, and Y. Luo, “Nomen est omen: exploring and exploiting similarities between argument and parameter names,” in *Proceedings of the 38th International Conference on Software Engineering, ICSE*, 2016, pp. 1063–1073.
- [18] B. Schwanke, *Survey of scope issues in programming languages*. Department of Computer Science, Carnegie-Mellon University, 1978.
- [19] S. Nguyen, C. T. Manh, T. K. Tran, T. M. Nguyen, T. Nguyen, K. Ngo, and H. D. Vo, “Arist: An effective API argument recommendation approach,” *J. Syst. Softw.*, vol. 204, p. 111786, 2023.
- [20] R. Scott, J. Ranieri, L. Kot, and V. Kashyap, “Out of sight, out of place: Detecting and assessing swapped arguments,” in *20th IEEE International Working Conference on Source Code Analysis and Manipulation*, 2020, pp. 227–237.

- [21] A. Habib and M. Pradel, “How many of all bugs do we find? a study of static bug detectors,” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 317–328.
- [22] R. Szalay, Á. Sinkovics, and Z. Porkoláb, “Practical heuristics to improve precision for erroneous function argument swapping detection in C and C++,” *J. Syst. Softw.*, vol. 181, p. 111048, 2021.
- [23] R. Bavishi, M. Pradel, and K. Sen, “Context2name: A deep learning-based approach to infer natural variable names from usage contexts,” *CoRR*, vol. abs/1809.05193, 2018.
- [24] M. A. Saied, A. Ouni, H. A. Sahraoui, R. G. Kula, K. Inoue, and D. Lo, “Improving reusability of software libraries through usage pattern mining,” *J. Syst. Softw.*, vol. 145, pp. 164–179, 2018.
- [25] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, “A statistical semantic language model for source code,” in *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE’13*, 2013, pp. 532–542.
- [26] M. Jimenez, C. Maxime, Y. Le Traon, and M. Papadakis, “On the impact of tokenizer and parameters on n-gram based code analysis,” in *Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 437–448.
- [27] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, “The daikon system for dynamic detection of likely invariants,” *Science of computer programming*, vol. 69, no. 1-3, pp. 35–45, 2007.
- [28] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall, “Analyzing apis documentation and code to detect directive defects,” in *Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 27–37.
- [29] H. Zhong, N. Meng, Z. Li, and L. Jia, “An empirical study on api parameter rules,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 899–911.
- [30] M. Pradel and K. Sen, “Deep learning to find bugs,” *TU Darmstadt, Department of Computer Science*, vol. 4, no. 1, 2017.
- [31] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, “Bug localization with combination of deep learning and information retrieval,” in *2017 IEEE/ACM 25th*

- International Conference on Program Comprehension (ICPC)*. IEEE, 2017, pp. 218–229.
- [32] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, “Deep learning similarities from different representations of source code,” in *Proceedings of the 15th international conference on mining software repositories*, 2018, pp. 542–553.
- [33] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, “Intellicode compose: Code generation using transformer,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, p. 1433–1443.
- [34] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, “On the naturalness of software,” *Communications of the ACM*, vol. 59, no. 5, pp. 122–131, 2016.
- [35] F. Liu, G. Li, Y. Zhao, and Z. Jin, “Multi-task learning based pre-trained language model for code completion,” in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 473–485.
- [36] X. Sun, X. Liu, J. Hu, and J. Zhu, “Empirical studies on the nlp techniques for source code data preprocessing,” in *Proceedings of the 2014 3rd international workshop on evidential assessment of software technologies*, 2014, pp. 32–39.
- [37] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, “Code-searchnet challenge: Evaluating the state of semantic code search,” *arXiv preprint arXiv:1909.09436*, 2019.
- [38] W. Wang, S. Shen, G. Li, and Z. Jin, “Towards full-line code completion with neural language models,” *arXiv preprint arXiv:2009.08603*, 2020.
- [39] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, “Language models are unsupervised multitask learners,” *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.
- [40] M. Izadi, R. Gismondi, and G. Gousios, “Codefill: Multi-token code completion by jointly learning from structure and naming sequences,” in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 401–412.
- [41] Y. Yang and C. Xiang, “Improve language modelling for code completion through learning general token repetition of source code.” in *SEKE*, 2019, pp. 667–777.
- [42] M. Bruch, M. Monperrus, and M. Mezini, “Learning from examples to improve code completion systems,” in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, 2009, pp. 213–222.

- [43] M. Ciniselli, N. Cooper, L. Pascarella, D. Poshyvanyk, M. D. Penta, and G. Bavota, “An empirical study on the usage of BERT models for code completion,” in *Proceedings of the 18th IEEE/ACM International Conference on Mining Software Repositories*, 2021, pp. 108–119.
- [44] V. Raychev, P. Bielik, and M. Vechev, “Probabilistic model for code with decision trees,” *ACM SIGPLAN Notices*, vol. 51, no. 10, pp. 731–747, 2016.
- [45] M. Allamanis, “The adverse effects of code duplication in machine learning models of code,” in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2019, pp. 143–153.
- [46] “Tokenizer for python source,” 2023. [Online]. Available: <https://docs.python.org/3/library/tokenize.html>
- [47] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” *arXiv preprint arXiv:1810.04805*, 2018.
- [48] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, “Codebert: A pre-trained model for programming and natural languages,” *arXiv preprint arXiv:2002.08155*, 2020.
- [49] Z. Tu, Z. Su, and P. Devanbu, “On the localness of software,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 269–280.
- [50] V. J. Hellendoorn and P. Devanbu, “Are deep neural networks the best choice for modeling source code?” in *Proceedings of the 2017 11th Joint meeting on foundations of software engineering*, 2017, pp. 763–773.
- [51] J. Li, Y. Wang, M. R. Lyu, and I. King, “Code completion with neural attention and pointer networks,” in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, 2018, pp. 4159–4165.
- [52] V. Raychev, M. T. Vechev, and E. Yahav, “Code completion with statistical language models,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, 2014, pp. 419–428.
- [53] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, “Deep code comment generation,” in *Proceedings of the 26th conference on program comprehension*, 2018, pp. 200–210.

- [54] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, “Improving automatic source code summarization via deep reinforcement learning,” in *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, 2018, pp. 397–407.
- [55] H. Wei and M. Li, “Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code.” in *IJCAI*, 2017, pp. 3034–3040.
- [56] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, “A novel neural source code representation based on abstract syntax tree,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 783–794.
- [57] M. Asaduzzaman, C. K. Roy, S. Monir, and K. A. Schneider, “Exploring api method parameter recommendations,” in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015, pp. 271–280.
- [58] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, “Exploring the limits of transfer learning with a unified text-to-text transformer,” *Journal of Machine Learning Research*, vol. 21, no. 140, pp. 1–67, 2020.
- [59] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, “Curating github for engineered software projects,” *Empirical Software Engineering*, vol. 22, pp. 3219–3253, 2017.
- [60] “Python abstract syntax grammar-ast module.” [Online]. Available: <https://docs.python.org/3/library/ast.html>
- [61] J. Bartlett, “Introducing functions and scope,” in *Programming for Absolute Beginners: Using the JavaScript Programming Language*. Springer, 2022, pp. 133–143.
- [62] Y. Wang, H. Le, A. D. Gotmare, N. D. Bui, J. Li, and S. C. Hoi, “Codet5+: Open code large language models for code understanding and generation,” *arXiv preprint arXiv:2305.07922*, 2023.
- [63] P. Delobelle, T. Winters, and B. Berendt, “Robbert: a dutch roberta-based language model,” *arXiv preprint arXiv:2001.06286*, 2020.
- [64] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, “Code llama: Open foundation models for code,” *arXiv preprint arXiv:2308.12950*, 2023.
- [65] D. K. Po, “Similarity based information retrieval using levenshtein distance algorithm,” *Int. J. Adv. Sci. Res. Eng.*, vol. 6, no. 04, pp. 06–10, 2020.

- [66] F. Rahutomo, T. Kitasuka, and M. Aritsugi, "Semantic cosine similarity," in *Proceedings of the 7th international student conference on advanced science and technology*, vol. 4, no. 1, 2012, p. 1.
- [67] V. U. Thompson, C. Panchev, and M. Oakes, "Performance evaluation of similarity measures on similar and dissimilar text retrieval," in *Proceedings of the 2015 7th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K)*, vol. 1, 2015, pp. 577–584.
- [68] N. Pradhan, M. Gyanchandani, and R. Wadhvani, "A review on text similarity technique used in ir and its application," *International Journal of Computer Applications*, vol. 120, no. 9, pp. 29–34, 2015.