DeepBugs for Python: Name-Based Bug Detection using Neural Networks

Manasa Sandhya Gunda *
msgunda@uwaterloo.ca

Sai Teja Ponugoti †
stponugo@uwaterloo.ca

1 Abstract

In the source code, elements like names of the functions and variables give a lot of information. Often this information conveys insights into the semantics constructed by the author, and therefore essential to understanding human programs. But this information is ignored by most of the present bug detectors thus missing some types of bugs. Some of the few existing [1] name-based bug detectors use the algorithms which are manually designed and fine tuned by finding the reasons on a syntactical level.

In this project, a name-based bug detection using learning the identifier names in Python files is proposed. In the proposed, our idea is to automatically learn about bugs by reasoning names based on semantic representation instead of manually writing them. This bug detection can be considered as a binary classification problem where the classifier is trained to differentiate between correct and incorrect code. This incorrect code will be created by changing the existing code with name-based bugs for training the bug detector.

2 Introduction

The problem addressed is Name-Based Bug Detection for Python code files. In the source code, since the identifier names contain valuable information, it can be used to detect defects missed by other bug detectors. In most of the existing bug detection tools, the defects due to identifier names are not found since it is challenging to reason these. The two main challenges involved in a name-based bug detector are the reasoning about the meaning of identifier names, and it should be able to analyze whether the given piece of code is correct or incorrect. If the model doesn't analyze properly, then it will lead to false positives while detecting actual bugs.

In software, a bug refers to an error, defect or flaw in any computer program or system. A bug creates unintended effects or triggers unpredictable actions on a machine. The method of detecting and fixing bugs is called debugging, and often tools and formal techniques are used to detect bugs. Some typos, particularly of symbols or logical/mathematical operators make the program operate incorrectly. In contrast, others, such as a missing symbol or miss spelt names, can prevent the program from running. Finding the former type of bugs, i.e. typos in symbols of logical and mathematical operators is very difficult to find. In order to find such bugs, the literal names can be used to obtain the context of the program, thus enabling bug detectors to find context-based bugs.This makes the learning problem interesting as the tool has to learn classification based on the natural language used by the programmer in the code.

Identifier names are mostly ignored in famous static analysis tools, such as FindBugs[3] and Google Error Prone[2].Current name-based bug detectors [4, 5] analysis rely on manually crafted

*Contributed equally
†Contributed equally

algorithms that use hard-coded patterns and carefully tuned heuristics to make decisions about programs, such as reporting a piece of code as likely to be incorrect. For example, an analysis focused on names that have been implemented recently by Google in 2017[5] has a set of heuristics to maximize the number of errors found and reducing the number of false positives. The design and fine-tuning of such heuristics impose a considerable amount of human effort.

In this proposal, with a machine learning-based approach, we want to tackle this problem of name-based bug detection. We use a learned vector representation of identifiers to address this reasoning problem to interpret the significance of identifiers in bug detection for Python code files. Our approach is inspired by DeepBugs [6], which was implemented for JavaScript files. We call our approach DeepBugs for Python.

3 Description

In this project, we try to deal with the problem of name-based bug detection with a machine learning-based approach. The identifiers are represented using a learned vector to handle the problem of reasoning about the meaning of the identifiers. This learned vectors, called embeddings, tries to preserve the semantic relatedness. For example the similarity between *count* and *length*. These embeddings were successful in various natural language processing tasks. Thus there is a need for adopting these embeddings in name-based bug detection. Binary classification is used to handle the problem of deciding whether code is correct or incorrect, and the model is trained for binary classification. Unlike the algorithms which need manual designing and tuning, this binary classifier learns without any human intervention.

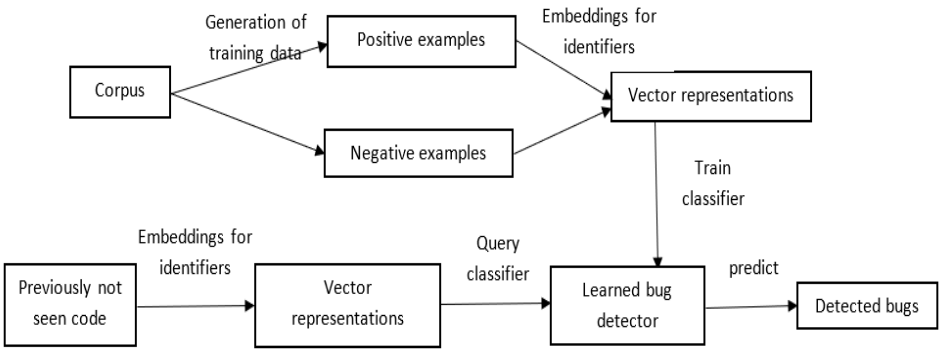

Figure 1: Overview of our strategy

We implement multiple types of name-related bugs. Firstly, we extract positive training examples from a code corpus then apply a simple transformation to get negative training examples. Then the binary classifier is trained to distinguish between these two. The trained model is then used for identifying mistakes in unseen code. In the referred paper[6], three types of bug detectors that find incorrect binary operators, accidentally swapped function arguments, and incorrect operands in binary operations are implemented. In this proposed paper, these same types will be implemented, but instead of JavaScript code files, we will implement for Python code files.

Figure 1 depicts the overview of out approach. All the steps and strategies are briefly discussed in the coming sections.

3.1 Embeddings for Identifiers and Literals

Since machine learning relies on vector representations of the analysed data, we need vector representations of code snippets to learn a bug detector. A significant challenge for a name-based bug detector is the rationale for identifier names, which are natural language information and are inherently difficult for a computer to understand.

Our goal is to distinguish between semantically similar identifiers and those of a different nature. For example, the bug detector searching for swapped arguments can learn from examples such as function_name(error, result) that function_name(res, err) is likely to be wrong, because error $\approx$ err and result $\approx$ res, where there is some reference to semantic similarity. Seq and sequoia, on the other hand, are semantically dissimilar in that they refer to different concepts, although they share a common prefix of characters. As illustrated by these examples, semantic similarity does not always correspond to lexical similarity, as considered in previous work [7, 8, 5], and cross-type boundaries might even exist. To enable a machine-based learning bug detector to reason about identifiers and their semantic similarities, we need to present identifiers that preserve these semantic similarities. In addition to identifiers, we also consider code literals such as "true" and "23" as they also convey relevant semantic information that can help detect bugs [6].

For each identifier based on a code corpus, DeepBugs for Python reasons about identifiers by automatically learning a vector representation called embedding. A naive representation like one-hot encoding also can be used to represent literals as vectors, but such a local representation does not allow DeepBugs for Python to generalise across non-identical but semantically similar identifiers, and we need an embedding that assigns a vector to semantically similar identifiers. Our distributed embedding is inspired by natural language word embedding, specifically by Word2Vec [9]. Word2Vec's basic idea is that the meaning of a word may be derived from the different contexts in which that word is used. We apply this concept to source code by treating code as a sequence of tokens and by describing the meaning of an identifier's occurrence as its tokens that immediately precede and follow. We used Word2Vec's CBOW version [9], which trains a neural network which predicts a token from the tokens around it. We represented each vector with a 200-dimensional vector and limited vocabulary to 20,000 most frequent identifiers. We trained word embeddings using gensim [10] Python library.

3.2 Vector Representations

Since code snippets are extracted from a corpus, our approach uses identifier embeddings to describe each snippet as a learning vector. Each bug detector built on top of these identifiers framework selects a code representation suitable for the particular code snippet type (explained in detail in following subsections). All bug detectors share the same technique in extracting expression names.

In this section, we present three examples of name-based bug detectors developed on top of the DeepBugs for Python (learned embeddings) system. The bug detectors address a variety of programming errors: accidentally swapped function arguments, incorrect binary operators, and incorrect binary expressions operands.

Each of these bug detectors is composed of two primary ingredients:

3.2.1 Generation of training data

Simple traversal of AST to generate training data that traverses the code corpus and extracts data of positive and negative code samples for a specific bug pattern based on a code transformation.

3.2.2 Code representation

Map each code sample to a vector for the Machine learning model to learn and classify it as either buggy or normal code snippet.

Three types of bug detectors that are implemented in this project are:

3.3 Swapped Function Arguments

First one of the implemented bug detectors is to identify accidentally swapped function arguments. This bugs can appear in both dynamic and statistic languages. In languages that are statistically typed, these kinds of errors can be seen when the functions accept more than one equally typed arguments. In the case of languages that are dynamically typed, the errors can be seen when the function has more than two arguments since there will not be any checking of types statically. For example, consider the example in Table 1:

Table 1: Example for Swapped function arguments

Buggy code	**Description**

`def divide(num1, num2):`
`  return num1 / num2`
`if choice == '4':`
`  print(num1, "/", num2, "=",`
`  divide(num2, num1))`
	As seen in the example the function arguments are exchanged, thus can cause error later on.

Generation of training data: In order to generate the training examples from the given corpus, for Swapped Function Arguments approach the code is traversed through the AST of each file in the corpus to get the function calls with two or more arguments.

For each function call with two or more arguments the following information is extracted:

- The name of the function call, *callee*.
- The first and second argument names, *arg1* and *arg2*.
- The name of the base object, *base* if method call exists otherwise an empty string.
- For the arguments which are literals, the types are defined as *argtype1* and *argtype2* otherwise an empty string.
- The parameter names of the function definition, *param1* and *param2*.

The argument types that are extracted are the built-in types in Python like "number","string", "boolean" which is *True* and *False*, "list", "tuple", "None", "var", "dict" and "unknown" for the types which are not defined. In all the extracted terms, *callee*, *arg1* and *arg2* are the mandatory terms. If these terms are undefined or unknown then this function call is ignored.

For each function call, the extracted positive training examples can be represented in the form of:

$$x_{pos} = (base, callee, arg1, agr2, argtype1, argtype2, param1, param2)$$

The negative training samples are collecting by interchanging the arguments as below:

$$x_{neg} = (base, callee, arg2, agr1, argtype2, argtype1, param1, param2)$$

These negative samples are created w.r.t to the positive training examples extracted.

Code representation: The extracted information x_{pos} and x_{neg} is in the form of strings which needs to be represented in the form of vectors. To represent all the names callee, arg1, arg2, base, param1 and param2 are found from the learned embeddings as discussed in the previous section. To represent the types argtype1 and argtype2, we define a map where each built-in types are assigned a vector. For example vector for "number" is randomly assigned as [1, 1, 0, 0, 1]. Based on these both x_{pos} and x_{neg} are represented in the form of vectors.

The next two bug detectors deal with the binary operators.

3.4 Incorrect Binary Operator

In Incorrect Binary Operator bug detector, the model will try to detect wrong binary operators. Consider the example, when $i <= x$ is present instead of $i < x$. These kind of mistakes are hard to find, but since the model is built with other data like operands, this can give valuable information about the bugs present.

Generation of train data: The training examples are extracted from the AST to get the below information:

- The names of left and right operands as *left* and *right*.
- The operator of binary operation as *op*.
- the types of left and right operands as *lefttype* and *righttype*. If the operands are not literals, then pass an empty string.
- The parent, *parent* and grand parent, *grandP* AST nodes of the binary operation AST node.

The nodes *left* and *right* are the mandatory terms and if these terms are undefined, then the binary operation node is not considered. The grandparent AST node is also considered because of its significance in some of the binary operations.

The positive extracted example looks like:

$$x_{pos} = (left, right, op, lefttype, rightype, parent, grandP)$$

The negative samples can be derived as below:

$$x_{neg} = (left, right, op', lefttype, rightype, parent, grandP)$$

where $op' != op$.

Code representation: Similar to the above bug detector, all the strings are converted to vectors and then concatenated to give a resultant vector. The AST nodes *parent* and *grandP* are converted to vector with a map, with each type of node represented with an 8 bit representation.

3.5 Incorrect Operand in Binary Operation

The last bug detector implemented is Incorrect Operand in Binary Operation which deals with mistakenly changed operands in any binary operation. An example discussed in Table 2.

Generation of train data: The same information, as discussed in the Incorrect Binary Operator bug detector is also used here. Positive examples extracted looks the same as the above bug detector:

$$x_{pos} = (left, right, op, lefttype, rightype, parent, grandP)$$

But the negative examples are created with interchanging of left or right operands.

$$x_{neg} = (left', right, op, lefttype', rightype, parent, grandP)$$

or

$$x_{neg} = (left, right', op, lefttype, rightype', parent, grandP)$$

Table 2: Example for Swapped function arguments

Buggy code	**Description**
`if variable > size :` `  variable.length = 2*size`	In the example it can be that in if condition *variable.length* should have been compared with the *size* instead of *variable* itself thus can cause an error .

The operands are randomly selected for the negative samples.
Code representation: Code representation is same as the above mentioned bug detector

3.6 Implementation[1] and Experimental setup

To learn word embeddings, tokens for the code snippets has to be generated. We have used the "tokenizer"[2] Python library to extract all tokens of the corpus and learned word embeddings using gensim [10] python library.

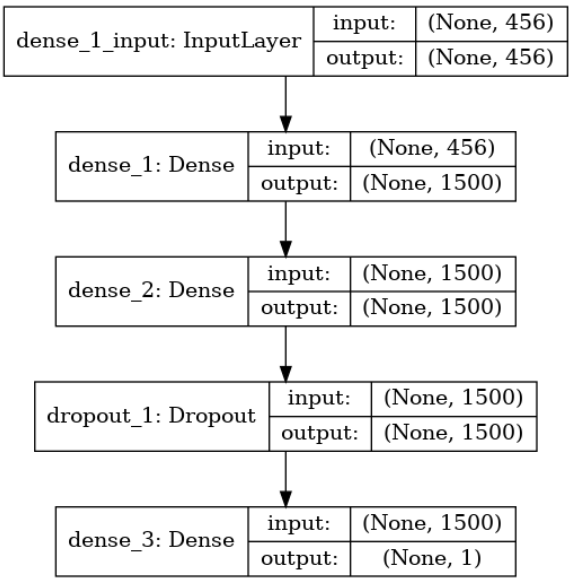

Figure 2: Neural Network model for Classification task.

Simple AST traversals based on the LARK[3], a Python parsing toolkit, are implemented to extract code and generate training examples. All collected data are written into text files by the Training data generator. The implementation of the bug detector, which builds on the TensorFlow and Keras[4] frameworks for deep learning, will then read those text files for loading data.

[1] https://github.com/sai-teja-ponugoti/DeepBugs-for-Python
[2] https://docs.python.org/3/library/tokenize.html
[3] https://github.com/lark-parser/lark
[4] https://www.tensorflow.org/guide/keras/sequential_model

We have used a Neural network with two-layers for classification task. The block diagram of model is as shown in Figure 2.

We use 150,000 Python files as a corpus of code generated by The Secure, Reliable, and Intelligent Systems Lab (SRI) [12]. The corpus includes files that were obtained from various open-source projects and were cleaned by deleting redundant files. For training, we use 100,000 python code files, and the remaining 50,000 files for testing the learned model. This is made sure for making model not to see the testing data. We only use the training files to learn the word embeddings and to determine the top 20,000 identifiers, which are part of our vocabulary.

All the above experiments mentioned are performed on a single machine with Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz, 2208 Mhz, 6 Core(s), 12 Logical Processor(s), 16GB of memory, an Intel(R) UHD Graphics 630 and a NVIDIA GeForce GTX 1050 Ti.

3.7 Training and validation Data Generation

Table 3 shows the training and testing data DeepBugs for Python extracts for the three bug detectors. Several thousands of training examples used to train each bug detector, which is large enough to train the classification models to achieve good accuracy. Half of the generated samples are positive, and the others are negative, so the dataset is quite balanced for the two classes. It would be impractical to manually build this amount of training data, including negative examples, this shows the value of our automated approach to data generation, which makes data generation quite efficient.

Table 3: Training and validation examples in different bug detectors implemented

Bug detector	**Training examples**	**Validation examples**
Swapped Arguments	196996	91344
Incorrect Binary Operator	231550	98570
Incorrect Binary Operand	133800	52576

3.8 Accuracy and True Positive Rate(TPR) of Bug Detectors

The following is a large scale assessment of the accuracy and recall of each bug detector based on automatically seeded bugs. Accuracy here informally means how many of the classification decisions the bug detector makes are correct. Recall means how many of the bugs the bug detector detects in a corpus of code. We train each bug detector on the 100,000 training Python files to evaluate these metrics and then apply it to the 50,000 validation Python files. We used the same procedure to generate positive and negative examples for validation, just as the one used for training data generation.

Table 4: True Positive rate, Training Accuracy and Validation Accuracy for all the bug detectors implemented with threshold as 0.5

Bug detector	**True Positive Rate(Recall)**	**Training Accuracy**	**Validation accuracy**	**Validation accuracy for positive samples**	**Validation accuracy for negative samples**
Swapped arguments	81.76	96.25	87.81	87.78	81.76
Incorrect Binary Operator	88.60	93.89	89.79	90.97	88.60
Incorrect Binary Operand	84.36	90.57	82.93	81.49	84.36

The Training accuracy of the bug detectors is, as shown in Table 4.

A bug detector's recall would be affected by how many warnings the detector can report. More the warnings, it is more likely to discover more bugs but also likely to report more false positives. In addition, developers are only able to review some warnings in operation. We assume, to calculate recall that all the warnings are inspected by a developer where the probability P(c)(probability of warning) is above a certain threshold. We convert this P into a boolean function to model this process:

$$P_t(c) = \begin{cases} 0, & \text{if } P(c) > t \\ 1, & \text{if } P(c) \leq t \end{cases} \tag{1}$$

The number of warnings to be reported is controlled by threshold t here. Using the above P(c), we compute recall as:

$$recall = \frac{|\{c \mid c \in C_{neg} \wedge P_t(c) = 1\}|}{h} \tag{2}$$

False positives are calculated using the below formula:

$$FP = |\{c \mid c \in C_{pos} \wedge P_t(c) = 1\}| \tag{3}$$

Note that both recall and count of false positives are estimates based solely on artificially seeded bugs. The measure of recall presupposes that all seeded bugs are actual bugs that should be detected. Here we consider these metrics because they help us to test the proposed framework with hundreds of thousands of artificial bugs, complementing the assessment with real-world bugs.

The recall of these three bug detectors for reporting warnings as a function of the threshold is shown in Figure 3. The results are shown for the threshold range $t \in [0.5,0.9]$. From Figure 3, we can notice that as the threshold increase, recall decreases, as the number of warnings given by bug detectors decreases. The results also show that some detectors of bugs are more likely to detect a bug than others if there is a bug.

True Positives, False Positives, True Negatives and False Negatives of the corresponding bug detectors are in the Table 5.

- True positive rate (TPR) : TP/(TP+FN)
- False negative rate (FNR) : 1-TPR
- True negative rate (TNR) : TN/(TN+FP)
- False positive rate (FPR) : 1-TNR

Table 5: True Positives, False Positives, True Negatives and False Negatives for different bug detectors implemented

Bug detector	**True Positives (TPR)**	**False Positives (FPR)**	**True Negatives (TNR)**	**False Negatives (FNR)**	**Total validation examples**
Swapped Arguments	40224 (81.76%)	5580 (12.22%)	40092 (87.78%)	5448 (18.24%)	91344
Incorrect Binary Operator	43668 (88.60%)	4448 (9.03%)	44837 (90.97%)	5617 (11.4%)	98570
Incorrect Binary Operand	22177 (84.36%)	4866 (18.51%)	21422 (81.49%)	4111 (15.64%)	52576

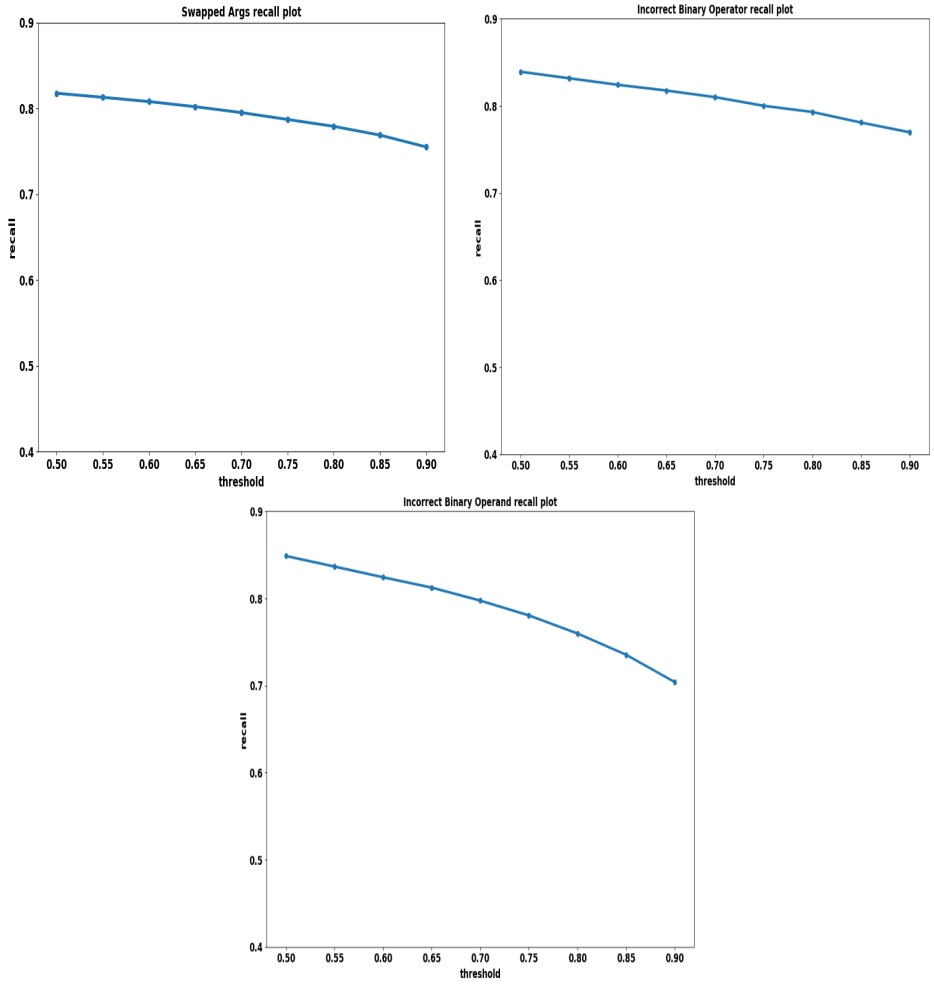

Figure 3: Recall plot for (a) Swapped Arguments (b) Incorrect Binary Operator (c) Incorrect Binary Operand

3.9 Efficiency of Bug detectors

The time taken to train a bug detector and use it for prediction on unseen code is shown in Table 6. The time shown in this table include both the time taken to extract the code example and then to train the bug detector or use it to predict previously unseen code. The time taken to both train and validate on all 150,000 Python files is a maximum of 38.6 minutes (for swapped arguments detector). The average prediction time taken for each Python file by all the three bug detectors is below 35ms, i.e. on an average 12ms per bug detector. From this, we can conclude that training time of the bug detectors is quite reasonable and once they are trained, they take very less time to debug the unseen code, i.e. within 0.36ms on an average.

- **Train examples extract time :** time to tokenize, traverse through AST and extract training examples from each file.
  - Number of files : 100k python code files
  - Average file size : 8.6 KB

Table 6: Training and validation data extraction and using bug detector time (in seconds)

Bug detector	**Training Data Extract (in sec) for 100k files**	**Training Model (in sec) (no.of examples)**	**Validation Data Extract (in sec) for 50k files**	**Validation Predict (in sec) (no.of examples)**
Swapped arguments	815	1044 (196996)	428	29 (91344)
Wrong Binary Operator	316	1195 (231550)	144	20 (98570)
Wrong Binary Operand	316	705 (133800)	144	6 (52576)

- **Training Model time :** time taken to train Neural Network model with layers each of 1500 neurons for 10 epochs on extracted trained examples. (number of examples mentioned in brackets)
- **Validation Data extract :** Similar to train sample extraction procedure.
  - Number of files : 50k python code files
  - Average files size : 8.6 KB
- **Validation prediction time :** Time taken to predict the validation samples(number of examples mentioned in brackets).

4 Conclusions and future works

This paper addresses the idea of identifying bugs based on names used by programmers using a machine learning based approach. Our bug detectors learned on DeepBugs for Python framework can distinguish bugs in code by reasoning about the meaning of identifier names used by humans in programs.

The success of this bug detectors is mainly because of the two essential approaches in this framework: the identifier word embeddings learned based on the semantic representation of names is beneficial in learning the relatedness of names used in code and the second is using simple code transformations of code to form artificial bugs lead in training bug detectors that are accurate for real-world bugs also. Unlike previous work on name-based bug identification, we are targeting a dynamically typed language (Python) where names convey information about the intended semantics of code, which is useful without static types. Applying our framework on three bug detectors built on top of it to a wide body of Python code files shows that the bug detectors have a testing accuracy of 82% to 90%.

We implemented three kinds of bugs, namely, incorrect binary operators, accidentally swapped function arguments, and incorrect operands in binary operations. The future works including implementing other types of common name-based bugs like incorrect assignment, missing arguments to functions call. This framework can experiment with different neural network architectures. Instead of a 2-layer Dense model used in this project Figure 2, further research can be done by using other Deep Learning models like CNN, LSTM, RNN which works very efficiently for Natural Language processing tasks.

In the long run, we hope our framework will assist in developing bug detectors for Python code files with minimal human effort and reduce the manpower needed to develop heuristics or even complement the existing bug detectors to improve their efficiency.

References

[1] Rice, A., Aftandilian, E., Jaspan, C., Johnston, E., Pradel, M., & Arroyo-Paredes, Y. (2017). Detecting argument selection defects. Proceedings of the ACM on Programming Languages, 1(OOPSLA), 1–22. https://doi.org/10.1145/3133928

[2] Aftandilian, E., Sauciuc, R., Priya, S., & Krishnan, S. (2012). Building Useful Program Analysis Tools Using an Extensible Java Compiler. 14–23. https://doi.org/10.1109/SCAM.2012.28

[3] Hovemeyer, D., & Pugh, W. (2004). Finding bugs is easy. 132–136. https://doi.org/10.1145/1028664.1028717

[4] Høst, E. W., & Østvold, B. M. (2009, July). Debugging method names. In European Conference on Object-Oriented Programming (pp. 294-317). Springer, Berlin, Heidelberg.

[5] Rice, A., Aftandilian, E., Jaspan, C., Johnston, E., Pradel, M., & Arroyo-Paredes, Y. (2017). Detecting argument selection defects. Proceedings of the ACM on Programming Languages, 1(OOPSLA), 1-22.

[6] Pradel, M., & Sen, K. (2018). DeepBugs: a learning approach to name-based bug detection. Proceedings of the ACM on Programming Languages, 2(OOPSLA), 1–25. https://doi.org/10.1145/3276517

[7] Liu, H., Liu, Q., Staicu, C. A., Pradel, M., & Luo, Y. (2016, May). Nomen est omen: Exploring and exploiting similarities between argument and parameter names. In Proceedings of the 38th International Conference on Software Engineering (pp. 1063-1073).

[8] Pradel, M., & Gross, T. (2011). Detecting anomalies in the order of equally-typed method arguments. ISSTA '11.

[9] Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781.

[10] RaRe Technologies. gensim: Topic Modelling for Humans, (GitHub repo). Last accessed June 15, 2020.

[11] Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. CoRR abs/1301.3781 (2013). http://arxiv.org/abs/1301.3781

[12] https://eth-sri.github.io/py150