A Partial Replication of "DeepBugs: A Learning Approach to Name-based Bug Detection"

Jordan Matthew Winkler
Purdue University
West Lafayette, Indiana, USA
jwinkler@purdue.edu

Abhimanyu Agarwal
Purdue University
West Lafayette, Indiana, USA
agarw184@purdue.edu

Caleb Tung
Purdue University
West Lafayette, Indiana, USA
tung3@purdue.edu

Dario Rios Ugalde
Lockheed Martin
Arlington, Texas, USA
dugalde@purdue.edu

Young Jin Jung
Purdue University
West Lafayette, Indiana, USA
jung199@purdue.edu

James C. Davis
Purdue University
West Lafayette, Indiana, USA
davisjam@purdue.edu

ABSTRACT

We replicated the main result of *DeepBugs* [4], a bug detection algorithm for name-based bugs. The original authors evaluated it in three contexts: swapped-argument bugs, wrong binary operator, and wrong binary operator operands. We only attempted to replicate the swapped-argument bugs. Following the DeepBugs algorithm described in the original paper, we independently implemented the major components from scratch: training set generation, token vectorization, and neural network data pipeline, model, and loss function. Using the same dataset and the same testing process, our implementation achieves comparable performance: within 2% of the accuracy reported by Pradel and Sen (P&S). Our replication artifact is available at: https://doi.org/10.5281/zenodo.5110820.

1 THE DEEPBUGS ALGORITHM

The DeepBugs algorithm is a means by which to identify *name-based bugs*. These bugs are cases where a developer uses the wrong symbols — *e.g.,* variable names — in a context, such as invoking a function with arguments in the wrong order. The DeepBugs algorithm supposes that developers choose good variable names that communicate meaning to their readers [2]. The DeepBugs algorithm uses a neural network to learn, from a large set of variable usages, the typical semantic meanings associated with usage contexts. Then, possible name-based bugs can be detected by identifying contexts where variable names are used inconsistently with the learned semantic meanings.

The DeepBugs algorithm components are illustrated in Figure 1.

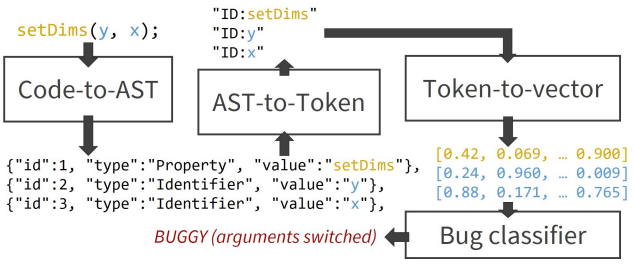

Figure 1: DeepBugs converts source code into Abstract Syntax Trees (ASTs), then to semantic-encoded vectors via Word2Vec. A neural network determines whether meanings match usage contexts. Here, a developer has written a function call for *setDims(width, height)* using *setDims(y, x)*. DeepBugs learns that *x* and *width* are semantically similar, as are *y* and *height*, so it predicts that the arguments are swapped.

Component	Shared dependencies
Code-to-AST	Acorn
AST-to-Token	*None*
Token-to-Vector	*None*
Bug Classifier	Tensorflow/Keras
Dataset	SRILab

Table 1: We independently implemented the DeepBugs components. We share the dependencies indicated in this table.

2 REPLICATION: DEEPBUGS COMPONENTS

Our implementation is based on the authors' description in the paper. See Table 1 for shared dependencies. Details follow.

Although the DeepBugs approach can be applied to any symbolic variable usage, and was evaluated in several contexts, our focus is on replicating the class of *swapped-argument bugs* — function arguments in the wrong order. The other bug classes follow from the success of this replication. Our explication is written accordingly.

2.1 Step 1: Code-to-AST

The original authors used the Acorn JavaScript parser [1] to convert JavaScript source code files into a JSON formatted Abstract Syntax Tree (AST). We converted the AST into a flat form with `id` and `children` AST attributes, which identifies the AST nodes that are acted upon by a given statement or call block. This allows us to swap the order of elements in the `children` list to generate swapped-argument bugs, without processing all nodes of a program.

2.2 Step 2: AST-to-Token

For a given bug pattern, only some nodes in the AST need be evaluated. Further, only some data from each node is semantically meaningful. For the swapped-argument bug pattern, we are only interested in 2-argument function calls, and only the function argument names thereof. To generate tokens, we derive tokens from each AST node's *value*, if present; else from its `children`. Our token generation rules are similar to those used by P&S:

- *CallExpression* with multiple `Identifier` children: tokenize child 1 as function name, the remainder as arguments.
- *MemberExpression* with `Identifier` and `Property` children: tokenize into object name and method/property names.
- Variable names and function names: tokenize as identifiers.
- JavaScript literals: tokenize as literal values.
- Other language elements (*e.g.,* conditional logic): ignore.

Our rules differ syntactically from the original authors' rules. Their rules apply to source code, while ours apply to ASTs, since we re-used their tokenized version of the SRILab dataset [6]. Using our rules, we crawled the dataset and tokenized each file's AST.

First, we extract every 2-argument function call group from the ASTs generated for the dataset. Specifically, we extract every function call that fits either of the following patterns:

- *functionName(arg1, arg)*
- *objectName.methodName(arg1, arg2)*

To create negative data samples, we swap the order of *arg1* and *arg2* for all extracted 2-argument function calls. Thus, our training and testing data is comprised of 2x the number of extracted samples.

2.3 Step 3: Token-to-Vector

To embed tokens as semantic vectors, Pradel & Sen trained a Collected Bag Of Words (CBOW) Word2Vec model [3]. Word2Vec can capture lexical similarities between two words, based on the context of the other words surrounding the two words. For example, variables *x* and *width* are not obviously similar. However, when developers use them in the context of *setDims()*, these variable names may take on similar meanings: "*width along x-dimension*". Word2Vec would thus embed them as nearby vectors.

To train our own CBOW Word2Vec model (we used the Gensim version [7]), we use our AST-to-Token extractor to list all tokens, in order of appearance, for a given JavaScript file's AST. The list of tokens is then used as a "paragraph" of words to train Word2Vec.[1]

2.4 Step 4: Bug Classifier

The DeepBugs bug classifier is a small neural network (Figure 2). Input passes through a 20% dropout regularization layer, a 200-node hidden layer with ReLU activations, another 20% dropout layer, and a final sigmoidal node that makes a buggy/not buggy prediction.

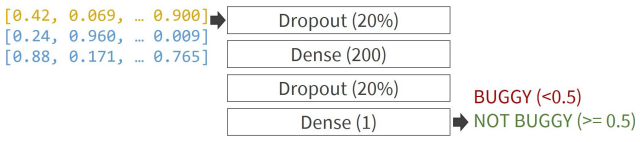

Figure 2: The DeepBugs algorithm uses a small neural network as a classifier for name-based bugs.

For the 2-argument swapped-argument bugs, the network input is the concatenation of the function name and argument vectors.

3 REPLICATION: EXPERIMENTS

Like the original authors, we evaluate our replicated DeepBugs using the 150K JavaScript Dataset [5]. The dataset is already partitioned into training and evaluation categories.

[1]P&S used a window size of 20 tokens. Due to clerical error we used a size of 200.

The DeepBugs algorithm is designed to detect any name-based bug pattern. The original authors evaluated it in three contexts: swapped-argument bugs, wrong binary operator, and wrong binary operator operands. We only considered swapped-argument bugs.

Each data sample is vectorized using the following procedure. Via the AST-to-Token method described in Step 2, we convert each data sample into a tuple of three tokens (e.g., (`"ID:funcName"`, `"ID:paramName"`, `"LIT:true"`)). Each of the three tokens is individually vectorized using the Word2Vec model we trained in Step 3. The final, classifier-ready sample is comprised of the three vectors, concatenated into a larger vector.

To train our DeepBugs classifier from Step 4, we use the vectorized data samples from the 150k JavaScript dataset's training partition. Training uses the P&S hyperparameters: 10 epochs, batch size 100, RMSprop optimizer with binary cross-entropy loss.

As shown in Table 2, our results are similar to those reported by P&S. There are minor differences. P&S claim to use a vocabulary size of "10,000" unique tokens. In actuality, the Word2Vec model they provide and the model we train both retrieve (the same) 9,994 tokens. We believe this difference was mere rounding. The vectors generated are also different, likely due to either (a) our clerical error in window size (see footnote 1), or (b) the different random number seed for Word2Vec. However, this difference does not appear to affect the core results. During step 4, we verify their method works by using the vectors as input to our DeepBugs model. Our accuracy is not exactly the same: although our training loss drops to roughly 0.002, like their model, our model performs with 2% worse accuracy.

Step	Comparison Metric	Original VS Replication
1 & 2	Token vocabulary size	Close ("10K" vs. 9,994)
	Extracted token values	Identical
3	Generated vectors	Close (window size; RNG seed)
4	Training Loss	Close: ~0.002 VS ~0.002
	Test Error	Close: ~0.04 VS ~0.06

Table 2: On the swapped-argument case from the 150k JavaScript Dataset, our DeepBugs replication successfully captured similar performance to the original authors' work.

4 DISCUSSION OF REPLICATION

Our implementation was simplified by the authors' use of an open-source JS dataset. The authors were generally diligent in reporting relevant information, but did omit one hyperparameter: RNG seeds.

Our implementation independently replicated the main results from the DeepBugs paper. We shared only the dataset (and P&S's mild postprocessing thereof). We therefore recommend that DeepBugs receive ACM's *Results Replicated* badge.

REFERENCES

[1] acornjs/acorn, May 2021. URL https://github.com/acornjs/acorn.
[2] Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. 2009.
[3] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space. 2013. URL http://arxiv.org/abs/1301.3781.
[4] Michael Pradel and Koushik Sen. DeepBugs: a learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages*, October 2018.
[5] Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. Learning programs from noisy data. *ACM SIGPLAN Notices*, 51(1):761–774, January 2016.
[6] Martin Vechev. 150k Javascript Dataset. URL https://www.sri.inf.ethz.ch/js150.
[7] Radim Řehůřek. Gensim: topic modelling for humans. URL https://radimrehurek.com/gensim/models/word2vec.html.