Detecting Performance Patterns with Deep Learning

Sophia Kolak
Columbia University
New York City, NY, USA
sdk2147@columbia.edu

ABSTRACT

Performance has a major impact on the overall quality of software projects. Performance bugs—bugs that substantially decrease run-time—have long been studied in software engineering, and yet they remain incredibly difficult for developers to handle. Because these bugs do not cause fail-stop errors, they are both harder to discover and to fix [5]. As a result, techniques to help programmers detect and reason about performance are needed for managing performance bugs. Here we propose a static, probabilistic analysis technique to provide developers with useful information about potential performance bugs at the statement level. Using leetcode samples scraped from real algorithms challenges, we use DeepWalk [6] to embed data dependency graphs in Euclidean space. We then show how these graph embeddings can be used to detect which statements in code are likely to contribute to performance bugs.

CCS CONCEPTS

• **Software and its engineering**; • **Computing methodologies** → *Machine learning*;

ACM Reference Format:
Sophia Kolak. 2020. Detecting Performance Patterns with Deep Learning. In *Proceedings of the 2020 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '20), November 15–20, 2020, Virtual, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3426430.3428132

1 INTRODUCTION

Performance is one of the most critical aspects of the software production platform. Inefficient software can quickly degrade user experience, cause unresponsive systems, and waste computing resources [3]. Even thoroughly tested software like Visual Studio and Microsoft SQLServer has had well documented and severe performance bugs [4]. Although many research tools exist to help combat these bugs, they remain both difficult to define and to detect.

This problem is only exacerbated in large software projects, where the precise code responsible for performance degradation can be incredibly small [3]. Gopstein et al. studied this phenomenon in the context of code perplexity [2]. By breaking code into its "atoms" of confusion, they were able to isolate the smallest patterns that confused developers.

In this work, we aim to detect performance patterns at a similarly atomic scale. We propose a graph-embedding of Python code that can help identify granular statements responsible for slowed performance. Using semantically equivalent but syntactically distinct solutions to algorithms challenges, we attempt to gain insight on the statistical patterns within graphs of source code that contribute to either high, or low performance.

```python
class Solution:
    def search(self, nums: List[int], target: int) ->
    ↪  bool:
        return target in nums
```

Figure 1: The shortest correct solution scraped from the leetcode problem "Search in a Rotated Sorted Array II" (high performing)

2 APPROACH

2.1 Data Mining

Leetcode[1] is an online platform for practicing algorithmic coding challenges designed to prepare software engineers for technical interviews. After a correct solution is submitted, leetcode provides a distribution of accepted solutions according to run-time, along with representative samples from other users along this distribution. We scraped all such available samples across 32 array problems, for a total of 1,836 code snippets. This allowed us to study real implementations of the same problem at variable run-times, and to isolate syntax as the cause of either high or low performance.

2.2 Graph Embedding

2.2.1 ***Abstract Syntax Trees.*** After obtaining the code samples, we used the Python script provided in the py150k data set[2] to create a serialized AST of each sample in JSON format. Because this script was designed for Python2 code and our samples were written in Python3, we first ran the 3to2[3] script on each solution. An example of the serialized JSON output is shown in Figure 2.

2.2.2 ***Data Dependency Graph.*** While the ASTs capture the program's control flow, they do not capture the data-flow, which often impacts performance. To account for data-flow in Python statically, we appended *data dependency* edges onto the preexisting

[1] https://leetcode.com/
[2] https://eth-sri.github.io/py150
[3] https://docs.python.org/3.0/library/2to3.html

SPLASH Companion '20, November 15–20, 2020, Virtual, USA

ACM ISBN 978-1-4503-8179-6/20/11…$15.00
https://doi.org/10.1145/3426430.3428132

```
[{"type":"Module","children":[1]},{"type":"ClassDef","
    children":[2,3,17],"value":"Solution"},{"type":"
    bases"},{"type":"body","children":[4]},{"type":"
    FunctionDef","children":[5,11,16],"value":"search"},
    {"type":"arguments","children":[6,10]},{"type":"args
    ","children":[7,8,9]},{"type":"NameParam","value":"
    self"},{"type":"NameParam","value":"nums"},{"type":"
    NameParam","value":"target"},{"type":"defaults"},{"
    type":"body","children":[12]},{"type":"Return","
    children":[13]},{"type":"CompareIn","children":[14,1
    5]},{"type":"NameLoad","value":"target"},{"type":"
    NameLoad","value":"nums"},{"type":"decorator_list"},
    {"type":"decorator_list"}]
```


Figure 2: An example of the serialized AST produced for the code sample shown in Figure 1

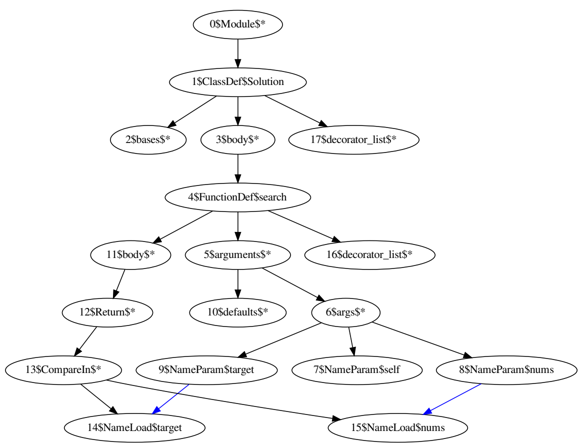

Figure 3: The AST for the code in Figure 1 and 2 with data dependency edges shown in blue.

ASTs. This allowed us to capture some aspects of data-flow without generating specific test cases or performing dynamic analysis.

In order to create these edges, we performed left to right depth first search on the AST. Whenever a node of type *NameStore* or *NameParam* was found, we created a stable entry in a dictionary with that node's value. Then, when the next *NameLoad* was encountered with the same value, we add an incoming data dependency edge from its parent load or store node and update the dictionary. Figure 3 shows an AST with its data dependency edges in blue.

2.2.3 ***DeepWalk***. DeepWalk [6] is an unsupervised deep learning technique that captures the social structure of graphs in Euclidean space. By performing random walks, DeepWalk estimates a high-dimensional distance between each pair of nodes. Given a graph, it encodes each node as an *n* dimensional vector, which can then be clustered and classified using traditional ML techniques.

In order to run DeepWalk, we indexed the nodes in each data dependency graph as integers and printed the structure as an adjacency list, meaning that no information about a node's type or value was explicitly passed to DeepWalk. After DeepWalk is run on a given sample, each node is encoded as a high-dimensional

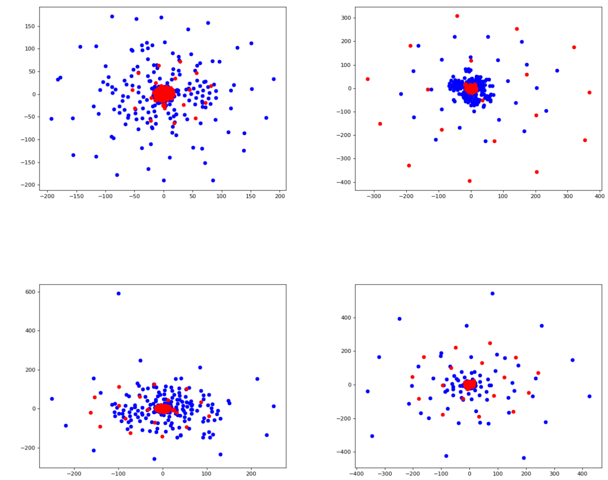

Figure 4: DeepWalk embeddings of high (red) and low (blue) performing samples on four runs.

vector. Finally, to visualize and better comprehend the results, we performed t-SNE [1] nonlinear dimensionality reduction to map each vector into $\mathbb{R}^2$.

3 PRELIMINARY RESULTS

To take an in-depth look at this embedding method, we compared our approach on low versus high performing code within one problem. We selected the problem "Search in Rotated Sorted Array II" because it was a medium difficulty problem with 19 solutions ranging from 104 to 32 ms. This was small enough for us to manually verify our embedding's correctness quickly, but also big enough to see the emergence of some general patterns.

After performing dimensionality reduction, each node in the graph is represented as a two dimensional vector. Figure 4 shows the result of plotting these vectors, where high performing samples (the top 50th percentile) are colored in red, and low performing samples (the bottom 50th percentile) are colored in blue. Because DeepWalk is stochastic, we created the embedding four times to ensure the results were consistent.

Interestingly, this embedding placed statements belonging to high performing samples near each other. At a high level, this means nodes of the data dependency graphs are more likely to be within a few hops. In contrast, many of the nodes in graphs of low performing samples were spread out more sporadically, meaning their nodes were less interconnected.

After classifying individual statements as low or high performance using this embedding, we can map the statements back onto the AST to determine which statements in code are most likely to be associated with low performance. This can give programmers useful information about which specific atoms in their code are likely to be limiting efficiency.

4 CONCLUSION

As this is only a preliminary analysis on a small dataset, we do not yet know how well these results generalize. Still, the initial findings suggest that deep learning and DeepWalk in particular may be useful for detecting the specific statements that contribute to performance bugs. We plan to continue scaling this methodology, and ultimately use this embedding to design a tool for automatically predicting low performance statements within large scale software projects.

REFERENCES

[1] Andrej Gisbrecht, Alexander Schulz, and Barbara Hammer. 2015. Parametric nonlinear dimensionality reduction using kernel t-SNE. *Neurocomputing* 147 (2015), 71–82.

[2] Dan Gopstein, Hongwei Henry Zhou, Phyllis Frankl, and Justin Cappos. 2018. Prevalence of Confusing Code in Software Projects: Atoms of Confusion in the Wild. In *Proceedings of the 15th International Conference on Mining Software Repositories*. ACM, 281–291. https://doi.org/10.1145/3196398.3196432

[3] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-World Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 77–88. https://doi.org/10.1145/2254064.2254075

[4] Microsoft. 2020. *Microsoft Docs*. https://docs.microsoft.com/

[5] A. Nistor, T. Jiang, and L. Tan. 2013. Discovering, reporting, and fixing performance bugs. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. 237–246.

[6] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: Online Learning of Social Representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '14)*. Association for Computing Machinery, New York, NY, USA, 701–710. https://doi.org/10.1145/2623330.2623732