Detecting Performance Patterns with Deep Learning

Sophia Kolak
Columbia University
New York City, 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. As a result, techniques to help programmers detect and reason about performance are needed for managing performance bugs. Here we propose a static, probabilistic embedding 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 to embed data dependency graphs in Euclidean space. We then describe 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 *Companion 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 [2]. Even thoroughly tested software like Visual Studio and Microsoft SQLServer has had well documented and severe performance bugs [3]. Although many research tools exist to help combat these bugs, they remain both difficult to define and to detect.

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

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

This problem is only exacerbated in large software projects, where the precise code responsible for performance degradation can be incredibly small [2]. Tools like JSNice, Nice2Predict, and DeGuard [5] successfully leverage machine learning to improve various aspects of programming, but a similar methodology has not yet been applied to performance.

In this paper, we begin building towards a tool that uses machine learning to infer inefficient statements in code. In doing so, one problem is finding a representation of the code which highlights differences in performance, and another is actually using that representation to produce a tool. Here we focus on only the former problem: using semantically equivalent but syntactically distinct solutions to algorithms challenges, we provide a graph-embedding which allows us to distinguish between low and high-performing statements.

2 Approach

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.

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

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

[1] https://leetcode.com/
[2] https://eth-sri.github.io/py150

```
[{"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":[14,15]},{"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

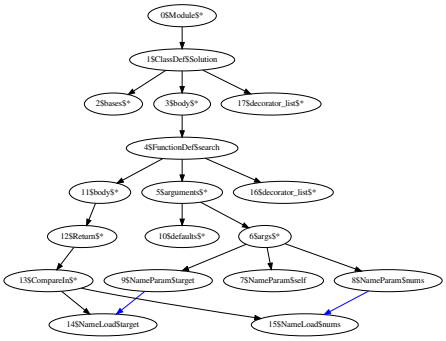

Figure 3. The AST for the code in Figure 1 and 2 with data dependency edges shown in blue.

Python3, we first ran the 3to2[3] script on each solution. An example of the serialized JSON output is shown in Figure 2.

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 ASTs. This allowed us to capture some aspects of data-flow without 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.

DeepWalk [4] 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 ML.

To run DeepWalk, we indexed the nodes in each data dependency graph as integers and passed the adjacency list

[3]https://docs.python.org/3.0/library/2to3.html

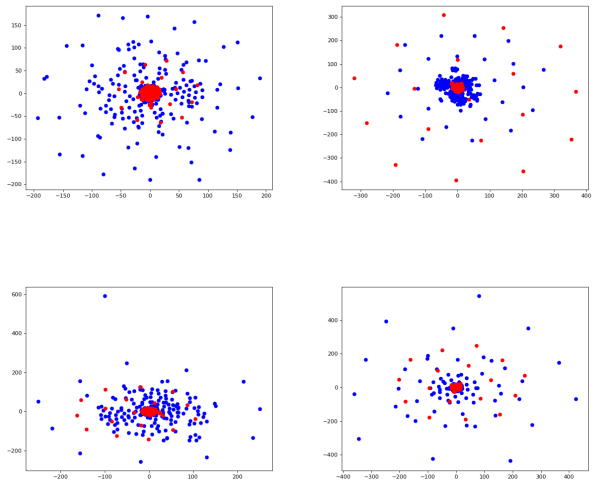

Figure 4. DeepWalk embeddings of high (red) and low (blue) performing samples on four runs.

to DeepWalk. After which each node is encoded as a high-dimensional 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 multiple times for consistency.

Interestingly, this embedding placed statements from high performing samples closer together than statements from low performing samples, meaning that the nodes in high-performing data dependency graphs were more likely to be within a few hops of each other.

Using this embedding, we plan to develop a classification method for the performance of individual statements within samples. After classifying statements as low or high performance, we can map the statements back onto the AST to determine which atoms of code are likely to be limiting efficiency.

References

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

[2] 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* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 77–88. https://doi.org/10.1145/2254064.2254075

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

[4] 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* (New York, New York, USA) *(KDD '14)*. Association for Computing Machinery, New York, NY, USA, 701–710. https://doi.org/10.1145/2623330.2623732

[5] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code". *SIGPLAN Not.* 50, 1 (Jan. 2015), 111–124. https://doi.org/10.1145/2775051.2677009