Code Runtime Complexity Prediction

1

Introduction

Confusion with Time Complexity Computation

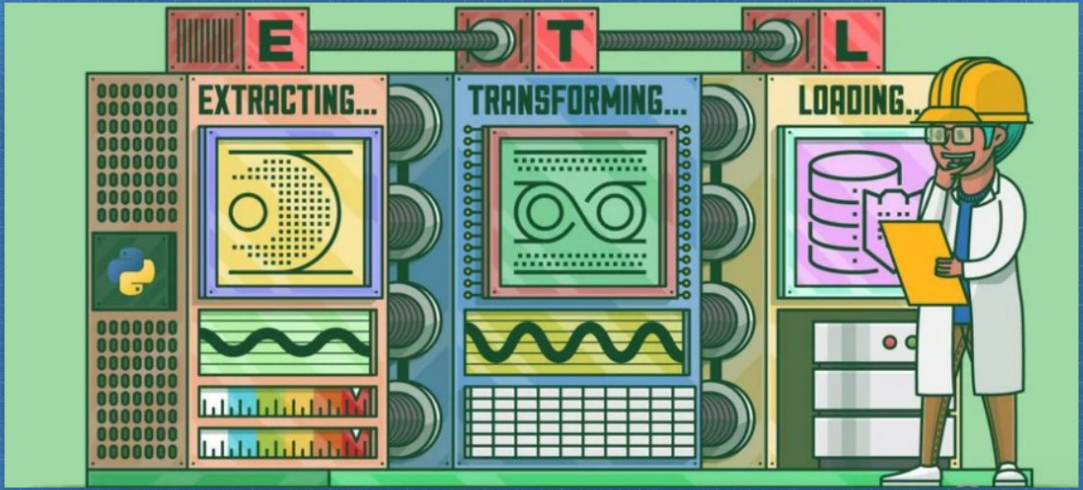
Confusion with Time complexity computation

Time complexity is the computational time taken by the particular algorithm to process the function of the input.

- Time complexity gets into the wrong way of consideration and gets diverted towards the execution time, which is incorrect because execution time is dependent on the hardware and is not ideal to be used as a standard measure to analyze the efficiency of the algorithm.
- Time complexity is the number of controlling elementary operations performed.
- To make it easier to compute, we regard the worst-case time complexity as the maximum amount of time required for inputs of a given size. Using big O notation, generally, O (n), O (n\log n), O (2^{n}), and so onwards., where n is the size in units of bits needed to represent the input.

Need of Machine Learning in complexity prediction

- Automated assessment of code submission on online platforms
- Analyses the code and lets the developers know how optimized their code is
- Can determine the complexity of all codes with polynomial order complexity



CONSTRUCTING DATASET

Dataset Extraction

To construct our dataset, we collected source codes of Java with distinct problems from Codeforces

For the construction of our dataset,

- We used the Codeforces API to retrieve problem and contest information
- used web scraping to download the solution source codes.
- Sampling of source codes is done based on DSA tags belonging to different complexity classes associated with the problem, e.g., binary search, sorting, etc.

Dataset correctness and validity

To ensure the correctness of evaluated runtime complexity, the source codes selected should be dry of issues, such as compilation errors and segmentation faults.

1. The codes that had the term Accepted or Time limit exceeded were the groups that were selected.
2. To ensure the accuracy of solutions, we only selected codes that passed at least four test input cases.
3. The criterion allowed us to include multiple solutions for a single problem, with different solutions having different runtime complexities.

Table 1. Classwise data distribution

From: Learning Based Methods for Code Runtime Complexity Prediction

Complexity class	Number of samples
$O(n)$	385
$O(n^2)$	200
$O(nlogn)$	150
$O(1)$	143
$O(logn)$	55

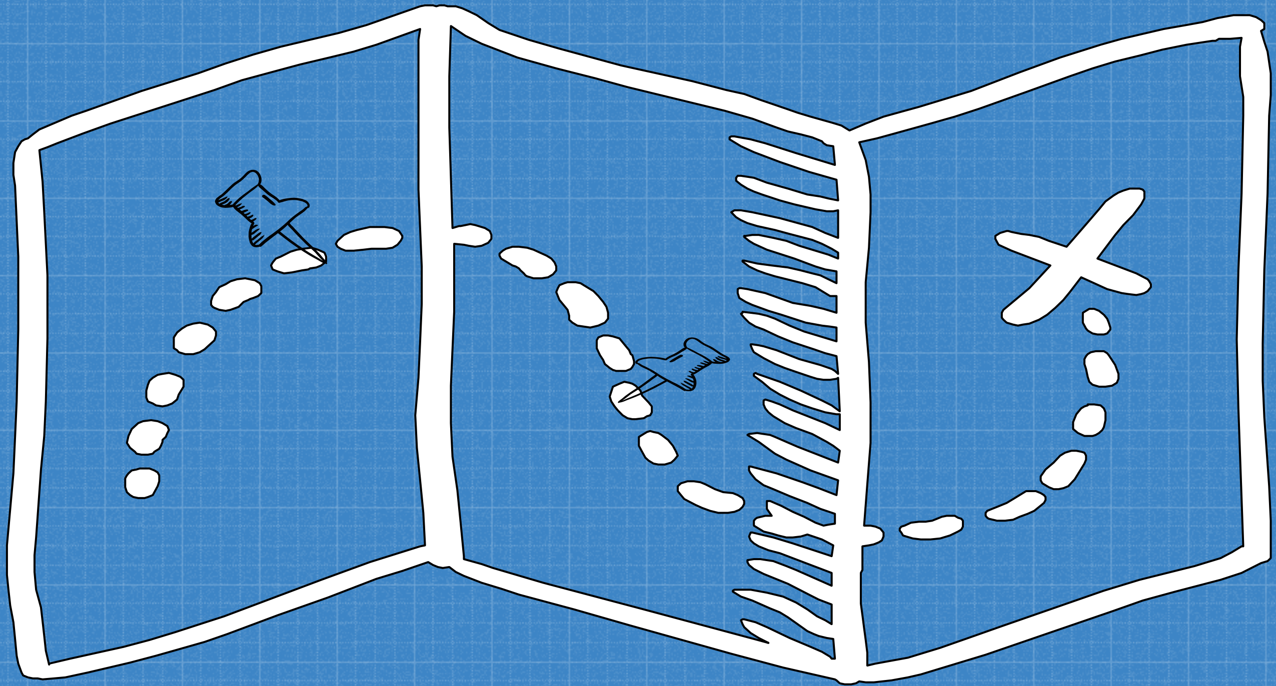
Table 2. Sample Extracted features

From: Learning Based Methods for Code Runtime Complexity Prediction

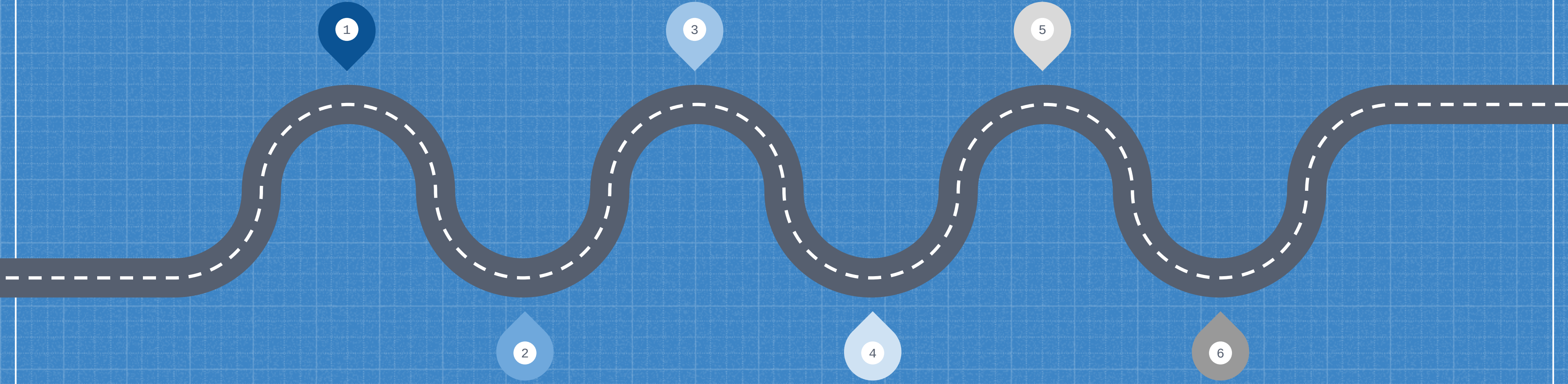
Features from code samples	
Number of methods	Number of breaks
Number of switches	Number of loops
Conditional-Loop frequency	Loop-conditional frequency
Loop-Loop frequency	Conditional-conditional frequency
Nested loop depth	Recursion present
Number of variables	Number of ifs
Number of statements	Number of jumps

We removed a few classes that didn't have sufficient data points and ended up with 932 source codes, 5 complexity classes, corresponding annotations, and extract features. The average number of problems per contest was 3. For 120 of these problems, we collected 4–5 different solutions, with different complexities.

Feature Engineering



ROADMAP of feature engineering



1. Identification of the features eg
Loops and conditions

2. Using AST to extract the features and using its depth first search to find syntax rules

3. Using JDT plugins to identify the unused code and removing them

4. Using the AST and using code embedding technique such as garph2vec

5. Graph2vec is analogous to doc2vec which predicts a document embedding given the sequence of words in it. The goal of graph2vec is, given a set of graphs G = {G1, G2, ...Gn}, learn a δ-dimensional embedding vector for each graph.

6. the AST representations, we used graph2vec to generate 1024-dimensional code embeddings. These embeddings were further used to train

Implementation of model

With a lesser amount of data and correctly hand-engineered features, Machine Learning (*ML*) methods outperform many *DL* models. Machine Learning (*ML*) methods are computationally less expensive compared to the latter. We compare the Multi-level Perceptron (MLP) classification to others and performed a similar analysis.

Table 3 depicts the accuracy score, weighted precision, recall and F1-score values for this classification task using 8 different algorithms, with the best accuracy score achieved using the ensemble approach of random forests.

Table 3. Accuracy Score, Precision and Recall values for different classification algorithms

Algorithm	Accuracy %	Precision %	Recall %	F1 score
K-means	50.76	52.34	50.76	0.52
Random forest	**71.84**	78.92	71.84	0.68
Naive Bayes	67.97	68.08	67.97	0.67
k-Nearest	65.21	68.09	65.21	0.64
Logistic Regression	69.06	69.23	69.06	0.68
Decision Tree	70.75	68.88	70.75	0.69
MLP Classifier	53.37	50.69	53.37	0.47
SVM	**60.83**	**67.62**	**67.00**	**0.65**

Table 4. Per feature accuracy score, averaged over different classification algorithms.

Feature	Mean accuracy
No. of ifs	44.35
No. of switches	44.38
No. of loops	51.33
No. of breaks	43.85
Recursion present	42.38
Nested loop depth	**62.31**
No. of Variables	42.78
No. of methods	42.19
No. of jumps	43.65
No. of statements	44.18

Limitations

- Our dataset is small compared to what is considered standard today.
- The moderate accuracy of the models is a limitation of our work.
- An important point to note is that using code embeddings is a better approach, still, their accuracy does not beat feature engineering significantly.
- One possible solution is to increase dataset size so that generated code embeddings can better model the characteristics of programs that differentiate them into multiple complexity classes when trained on a larger number of codes.

Conclusion

- The dataset presented and the baseline models established should serve as guidelines for future work in this area.
- The dataset presented is balanced and well-curated.
- The baselines present Code Embeddings and Handcrafted features have comparable accuracy, we have established through data ablation tests that code embeddings learned from the Abstract Syntax Tree of the code better capture relationships between different code constructs that are essential for predicting runtime complexity.
- Work can be done in the future to increase the size of the dataset to verify our hypothesis that code embeddings will perform significantly better than handcrafted features.

THE END