Big Code Analysis

Wei Le

April 5, 2021

Big code

- open source software systems: such as Linux, MYSQL, Django, Ant and OpenEJB
- source code and software development data: changes, commits ...
- *big code*: billions of tokens of code and millions of instances of data
- data-driven: combining machine learning and statistics
  - calculate statistical distributional properties
  - probabilistic models of source code
  - code embedding training

Industry and Academia Interests

- DARPA: Mining and Understanding Software Enclaves (MUSE)
- Startup: Deep Code
- Microsoft and ABB: using data to drive software development decisions

Applications: recommender system

Based on the language model built from copra and based on the current context of code, comments ...

- code completion: API sequence, any code
- comment completion
- improve the names of methods

Applications: translations

- Java programs to C and vice versa
- code to text/pseudo code
- text to code

Applications: mining patterns

- loop idioms
- code convention, anomaly code
- code clone and repetitive code

Applications: information retrieval and code understanding

- code search
- semantic classification: classify software based on their functionalities

Code Emebdding

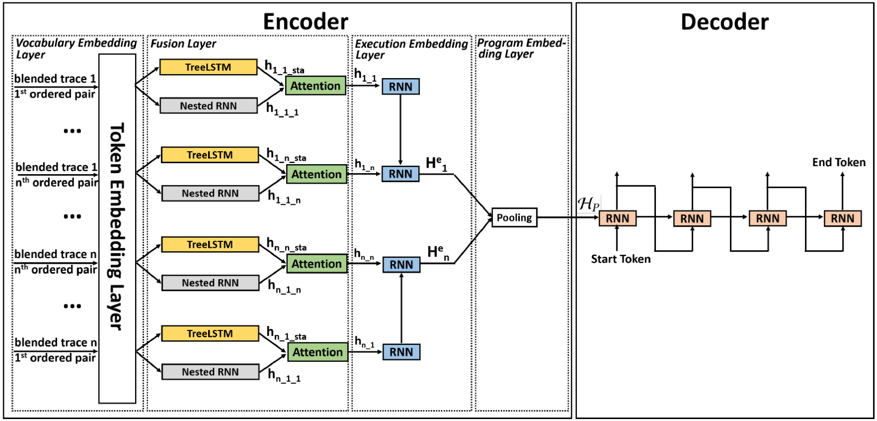
- Neural program embeddings: represent programs in vectors, each method/code snippet turns into a fixed-length vector
- Goal: enable neural network to solve problems for programs
- Input: source code (code2seq, code2vec), traces (DYPORO), graph representations of code such as AST, CFG and PDG (Devin), a mixture of code and traces, a mixture of source code and natural language, the output from lightweight code analysis
- Output: vector representation
- Requirement: efficient and precise
- Approaches: supervised, task specific

Code Embedding + Deep Learning Have Been Applied

Learning the information from existing code and help the tasks of new given code:

- detecting vulnerabilities: Devin @ NIPS 2019
- generating comments, code summarization: joint modeling of source code and NL @ NIPS 2015
- method name prediction: @2020 PLDI
- semantic classification (which algorithm/task accomplished?): @2020 PLDI
- code search
- clone detection
- repair student assignments from MOOC

Blended, Precise Semantic Program Embeddings



Blended, Precise Semantic Program Embeddings

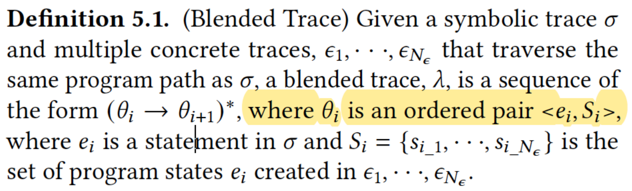
Overall architecture of the neural network for the two tasks: name prediction and semantic classification

- encoder: from programs to a vector H_p (embedding)
- decoder: from H_p to the predicted name
- linear transformation layer + softmax layer: predict which tasks/algorithms the program implements

Applying program analysis to get an intermediate product

Requirement: representing program semantics

- symbolic traces: for each method, used 20 symbolic traces
- testing the program to obtain the concrete traces: using Randoop unit test input generation, each symbolic trace will be exercised using 5 concrete inputs



Definition 5.1. (Blended Trace) Given a symbolic trace σ and multiple concrete traces, $\epsilon_1, \cdots, \epsilon_{N_\epsilon}$ that traverse the same program path as σ, a blended trace, λ, is a sequence of the form $(\theta_i \rightarrow \theta_{i+1})^*$, where θ_i is an ordered pair $<e_i, S_i>$, where e_i is a statement in σ and $S_i = \{s_{i\_1}, \cdots, s_{i\_N_\epsilon}\}$ is the set of program states e_i created in $\epsilon_1, \cdots, \epsilon_{N_\epsilon}$.

Encoding blended trace

Encoding a pair in the blended trace:

- encoding tokens to numbers: 9,641 unique tokens collected from the corpoa of programs, each will be encoded in a 100-dimension vector
- encoding statement: using TreeLSTM to compose tokens into a statement based on the AST
- encoding program state: support both values of primitive types and object types; using RNN
- combining the statement and values: using attention to learn a vector that combines the values and program statements

$$h_{i\_j} = \alpha_{i\_j\_sta} * h_{i\_j\_sta} + \alpha_{i\_1\_j} * h_{i\_1\_j} + \alpha_{i\_2\_j} * h_{i\_2\_j}$$

Encoding blended trace

Encoding the blended trace and program of blended traces

- Connect the odered pairs into a trace: RNN
- Combine all the traces into one embedding for program: pooling layer

Experimental studies - big code dataset

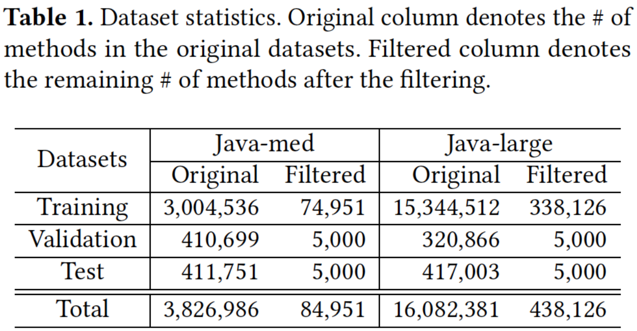
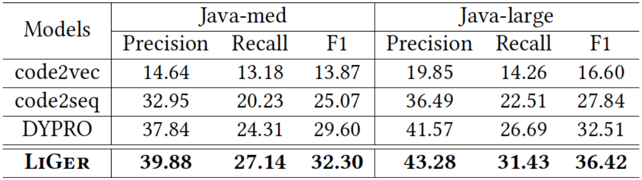


Table 1. Dataset statistics. Original column denotes the # of methods in the original datasets. Filtered column denotes the remaining # of methods after the filtering.

Datasets	Java-med		Java-large	
	Original	Filtered	Original	Filtered
Training	3,004,536	74,951	15,344,512	338,126
Validation	410,699	5,000	320,866	5,000
Test	411,751	5,000	417,003	5,000
Total	3,826,986	84,951	16,082,381	438,126

Experimental studies -name prediction results



Models	Java-med			Java-large		
	Precision	Recall	F1	Precision	Recall	F1
code2vec	14.64	13.18	13.87	19.85	14.26	16.60
code2seq	32.95	20.23	25.07	36.49	22.51	27.84
DYPRO	37.84	24.31	29.60	41.57	26.69	32.51
LiGer	**39.88**	**27.14**	**32.30**	**43.28**	**31.43**	**36.42**

$$\text{Precision} = \frac{tp}{tp + fp}$$

$$\text{Recall} = \frac{tp}{tp + fn}$$

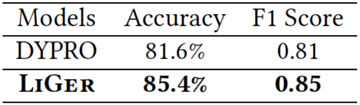
Precision: among all the messages you reported, how many are good messeges?
Recall: among all the messages you should report, how many are actually reported?

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} = \frac{\text{TP}}{\text{TP} + \frac{1}{2}(\text{FP} + \text{FN})}$$

Experimental studies - semantic classification results

- dataset: COSET, 85K programs developed by a large number of programmers while solving ten coding problems, algorithms
- 45,596 training programs, a validation set of 9,000 programs and a test set with the remaining 9,000
- architecture: modified with linar transformation + softmax regression

Table 3. LiGer's results on COSET.

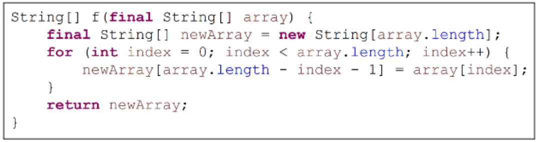


Models	Accuracy	F1 Score
DYPRO	81.6%	0.81
LiGer	**85.4%**	**0.85**

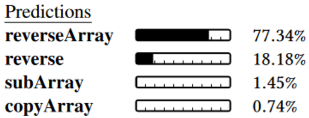
Ablation Studies

- After removing the static feature dimension, LiGer exhibits almost the same prediction accuracy (31.16 on Javamed and 35.21 on Java-large in F1 score).
- Removing dynamic feature has a much larger impact on LiGer's precision (20.23/22.95 on Java-med/Java-large).
- Removing attention has a notable impact on LiGer, which drop its F1 score from 32.30 (36.42) to 28.63 (33.71) on Javamed (Java-large).

Code2vec: Learning Distributed Representations of Code



```
String[] f(final String[] array) {
    final String[] newArray = new String[array.length];
    for (int index = 0; index < array.length; index++) {
        newArray[array.length - index - 1] = array[index];
    }
    return newArray;
}
```



- Training data: code snippets, and their labels
- Goal: predicting a name of a method (semantic labeling of code snippets)
- Demo: https://code2vec.org/

Overall Approach

1. obtain a path from ast
2. map a bag of paths to a real-value vector
3. learn the model
4. predict the name based on the model and the vector

Obtain an *AST path*

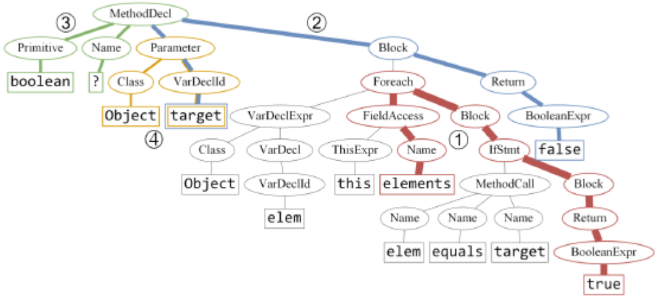
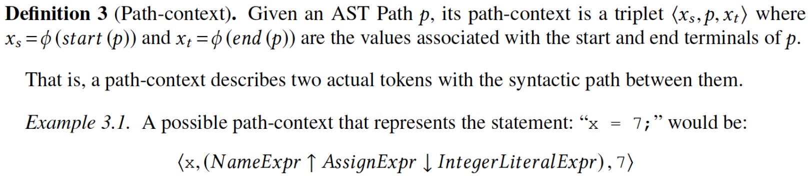


Fig. 3. The top-4 attended paths of Figure 2a, as were learned by the model, shown on the AST of the same snippet. The width of each colored path is proportional to the attention it was given (red ①: **0.23**, blue ②: **0.14**, green ③: **0.09**, orange ④: **0.07**).

$\langle$`elements`, Name$\uparrow$FieldAccess$\uparrow$Foreach$\downarrow$Block$\downarrow$IfStmt$\downarrow$Block$\downarrow$Return$\downarrow$BooleanExpr, `true`$\rangle$

Distributed Representation of a Method

- a bag of path-contexts for a method



Definition 3 (Path-context)**.** Given an AST Path p, its path-context is a triplet $\langle x_s, p, x_t \rangle$ where $x_s = \phi(start(p))$ and $x_t = \phi(end(p))$ are the values associated with the start and end terminals of p.

That is, a path-context describes two actual tokens with the syntactic path between them.

Example 3.1. A possible path-context that represents the statement: "`x = 7;`" would be:

$$\langle \mathtt{x}, (NameExpr \uparrow AssignExpr \downarrow IntegerLiteralExpr), \mathtt{7} \rangle$$

- the weight is initialized with random values, then learned from the training data

Neural Network Architecture

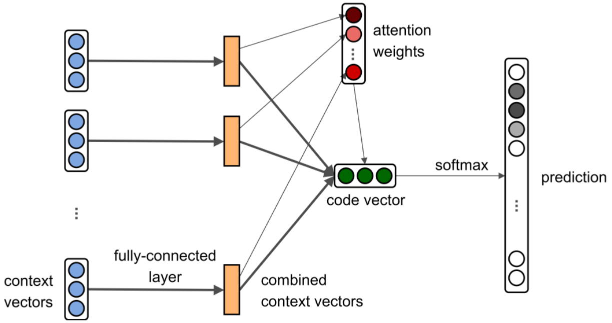
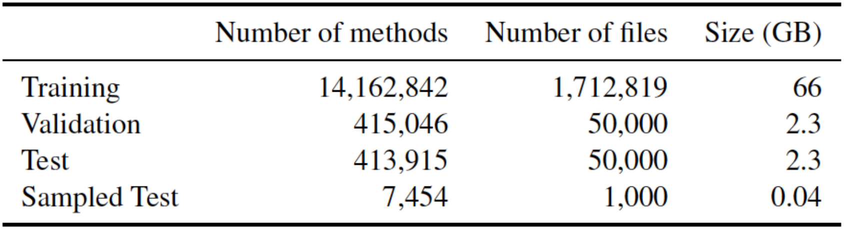


Fig. 4. The architecture of our path-attention network. A *full-connected layer* learns to combine embeddings of each path-contexts with itself; attention weights are learned using the combined context vectors, and used to compute a *code vector*. The code vector is used to predicts the label.

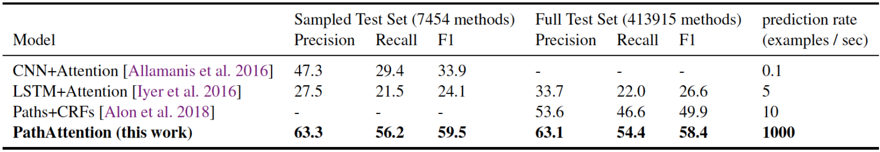
Experimental Evaluation - Dataset



	Number of methods	Number of files	Size (GB)
Training	14,162,842	1,712,819	66
Validation	415,046	50,000	2.3
Test	413,915	50,000	2.3
Sampled Test	7,454	1,000	0.04

Table 2. Size of data used in the experimental evaluation.

Experimental Evaluation - Results



	Sampled Test Set (7454 methods)			Full Test Set (413915 methods)			prediction rate
Model	Precision	Recall	F1	Precision	Recall	F1	(examples / sec)
CNN+Attention [Allamanis et al. 2016]	47.3	29.4	33.9	-	-	-	0.1
LSTM+Attention [Iyer et al. 2016]	27.5	21.5	24.1	33.7	22.0	26.6	5
Paths+CRFs [Alon et al. 2018]	-	-	-	53.6	46.6	49.9	10
PathAttention (this work)	**63.3**	**56.2**	**59.5**	**63.1**	**54.4**	**58.4**	**1000**

Table 3. Evaluation comparison between our model and previous works.

Probabilistic models of code: How to model big code via machine learning

- *generative models*: defines a probabilistic distribution over code by stochastically modeling the generation of smaller and simpler parts of code, e.g., what is the probability of deriving to this sub-tree as the left child of the root
- *representational models*: conditional probability distribution over code element properties, e.g., what is the probability of types X, Y and Z for this variable a (based on, e.g., surrounding context of a)
- *pattern mining models*: unsupervised learning to infer a latent (not observable) structure within code

Generative Models

How to generate code in different context:

- Input: training data D
- A output code representation: c
- A possibly empty context: C(c)
- Probabilistic distribution: $P_D(c|C(c))$

- language model: $C(c) = \emptyset$
- code-generative multimodal model: C(c) is a non-code modality, e.g., natural language
- transducer model: C(c) is also code

Generative Models

How to represent the code in generative models

- "a bag of words": a set of tokens
- a sequence of tokens: n-gram
- abstract syntax trees
- graphs

Structure prediction: Predicting program properties from big code

```
function chunkData(e, t) {
  var n = [];
  var r = e.length;
  var i = 0;
  for (; i < r; i += t) {
    if (i + t < r) {
      n.push(e.substring(i, i + t));
    } else {
      n.push(e.substring(i, r));
    }
  }
  return n;
}
```

(a) JavaScript program with minified identifier names

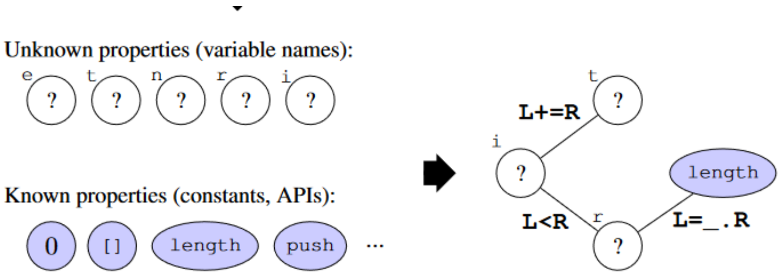
```
/* str: string, step: number, return: Array */
function chunkData(str, step) {
  var colNames = []; /* colNames: Array */
  var len = str.length;
  var i = 0; /* i: number */
  for (; i < len; i += step) {
    if (i + step < len) {
      colNames.push(str.substring(i, i + step));
    } else {
      colNames.push(str.substring(i, len));
    }
  }
  return colNames;
}
```

(e) JavaScript program with new identifier names and types

Predicting program properties from big code

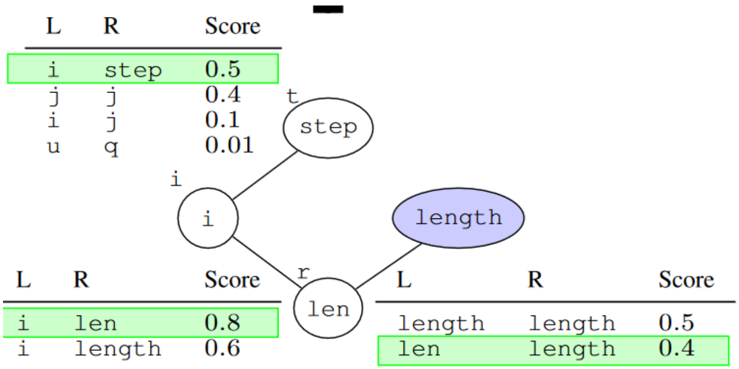
- code as a variable dependence network
- represent each variable as a node, the property of some node we know while the property of some node we don't know
- represent variable interaction as *conditional random field (CRF)* – a type of probabilistic graphical models that represent the conditional probability of lables y given observations of x $P(y|x)$
- train CRF to predict names and types of variables for Javascripts

Predicting program properties from big code


Unknown properties (variable names):
e
t
n
r
i
?
Known properties (constants, APIs):
0
[]
length
push
...
t
L+=R
i
?
length
L<R
r
L=_.R

(b) Known and unknown name properties (c) Dependency network

Predicting program properties from big code



L	R	Score
i	step	0.5
j	j	0.4
i	j	0.1
u	q	0.01

L	R	Score
i	len	0.8
i	length	0.6

L	R	Score
length	length	0.5
len	length	0.4

Pattern mining models

- Pattern mining models aim to discover a finite set of human-interpretable patterns from source code, without annotation or supervision
- Probabilistic pattern mining models of code infer the likely latent structure of a probability distribution

Pattern mining models

- Examples of pattern mining models: frequent pattern mining – it is not a probabilistic model but counting based
- Probabilistic model can find interesting models, sometimes frequent occurred patterns may not be interesting
- Most probably grouping of API elements

Summary

- Program analysis to get and represent information from code
- Code embedding: represent code using vectors (typically for certain tasks)
- Machine learning models/probabilistic models for big code
  - generative models (language models, bimodal, transducer model dependent on information available)
  - representation models: CRF (graphical model inference), distribution vector(encoding used by deep learning)
  - pattern mining models: clustering algorithms
- Applications:
  - predicting developers intent: recommender system
  - predicting anomaly code
  - code translation
  - clone and idiom mining
  - predict names
  - ...

Further Readings

- Learning to represent programs with graphs, ICLR 2018
- Code2seq: Generating sequences from structured representations of code, ICLR 2019
- Path-based Function Embedding and Its Application to Error-handling Specification Mining, FSE 2018
- Code Vectors: Understanding Programs Through Embedded Abstracted Symbolic Traces, FSE 2018
- Dynamic Neural Program Embedding for Program Repair, ICLR 2018
- Code2Vec: Learning Distributed Representations of Code, POPL 2019
- Blended, Precise Semantic Program Embeddings, PLDI, 2020
- COSET: A Benchmark for Evaluating Neural Program Embeddings, arXiv, 2019
- Predicting program properties from "big code"
- Deep Code
- Machine Learning for Programming
- A Survey of Machine Learning for Big Code and Naturalness (2018)
- Machine learning on source code