Code2vec: Learning Distributed Representations of Code

URI ALON, Technion, Israel
MEITAL ZILBERSTEIN, Technion, Israel
OMER LEVY, Facebook AI Research, USA
ERAN YAHAV, Technion, Israel

Source code at https://github.com/tech-srl/code2vec
Overview

• Motivating use-case: Present an extreme summarization of source code snippets into short, descriptive function name.
• Architecture: Attention model over path-contexts
• Design Choices: Focus on syntax level
• Evaluation: Dependent on variable naming
• Discuss Limitations: Tendency to pick up on coding convention over learned semantic meaning
• Online Demo: as time permits
Motivating Example: Semantic Method Naming

Try to predict attribute a meaningful name to methods based on the body
Motivating Example: Semantic Method Naming

Try to predict attribute a meaningful name to methods based on the body

```java
String[] ______(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;
}
```
Motivating Example: Semantic Method Naming

Try to predict a meaningful name to methods based on the body

```java
String[] reverseArray(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;
}
```

- `reverseArray`: 77.34%
- `reverse`: 18.18%
- `subArray`: 1.45%
General Task: Representation Learning

Raw Data → Feature Generation \( \frac{z_i + \alpha}{N + \alpha d} \) → Feature Representation → Model → Training Data

Learning Representations

Prior Knowledge

\[
\begin{align*}
0.21 \\
1.01 \\
0.01 \\
0.05 \\
0.37 \\
1, 0.5 \\
0, 0.1 \\
0, 0.0 \\
0, 0.2
\end{align*}
\]
History of Static Code Representation

**Exact Representation**

**Constructed Features**

**Deep Learning Features**

Static Rule Inference + Checking

**Binary Feature Vectors, N-Grams**

"Semantic Space" Vector Embeddings (code2vec)


Comparison with inst2vec

• ‘How far a syntactic-only approach can go’
• Purely syntactic
  • no control flow/data flow information
• Scales more effectively
  • 1K method per second training on over 12M methods
• Can interpret how predictions are reached
  • Using attention
A naïve approach: N-gram model

Variable: \( \text{isDone} \)

Features:
- boolean, in:MethodBody, final

Contexts:
- final boolean isDone = false;
- \( C_{-2}r_{\text{final}} + C_{-1}r_{\text{boolean}} + C_1r_{=} + C_2r_{\text{false}} \)
- while (!isDone) {
  - \( C_{-2}r_{=} + C_{-1}r_{=} + C_1r_{=} + C_2r_{=} \)

\[ s_\theta(.) = \mathbf{f}_\text{context}^T \mathbf{q}_{\text{isDone}} + b_{\text{isDone}} \]
Path Context: Example – Contains function

```java
boolean f(Object target) {
    for (Object elem: this.elements) {
        if (elem.equals(target)) {
            return true;
        }
    }
    return false;
}
```
Path Context: Parse into Abstract Syntax Tree
Path Context: representing the path

(elements, Name↑FieldAccess↑Foreach↓Block↓IfStmt↓Block↓Return↓BooleanExpr, true)
Path-context vector

(elements, Name ↑ FieldAccess ↑ Foreach ↓ Block ↓ IfStmt ↓ Block ↓ Return ↓ BooleanExpr, true)

One hot lookup table

One hot lookup table

One hot lookup tables

Concatenate

Fully Connected Layer

Path-context Vector: D = 200
Neural Network Architecture
Code2vec Architecture

Program → Bag of contexts (token1, path, token2) → Decompose → Aggregate → Predict

- **Decompose**: $\text{tanh}(Wx)$
- **Aggregate**: code vector, attention weights
- **Predict**: target vectors, prediction, softmax
Design Choices

- Bag of contexts
  - Existence not order

- Syntax-only

- Large corpus, simple model
  - “95% of the paths in the test set were already seen in the training set”
  - 1000 methods per second
  - 1.5 days to completely train a model

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

<table>
<thead>
<tr>
<th></th>
<th>Number of methods</th>
<th>Number of files</th>
<th>Size (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training</td>
<td>12,636,998</td>
<td>1,712,819</td>
<td>30</td>
</tr>
<tr>
<td>Validation</td>
<td>371,364</td>
<td>50,000</td>
<td>0.9</td>
</tr>
<tr>
<td>Test</td>
<td>368,445</td>
<td>50,000</td>
<td>0.9</td>
</tr>
<tr>
<td>Sampled Test</td>
<td>7,454</td>
<td>1,000</td>
<td>0.04</td>
</tr>
</tbody>
</table>
Evaluation

• Prediction of names compared to recent approaches
• Compare with no attention or hard attention
• Evaluate the relative contribution from the components of the context vector
• Interpreting code vectors
Evaluation Metric

• Case-insensitive Sub-token Matching
• countLines == linesCount
• countLines vs count
  • Full precision -> no false positives
  • Low recall -> has false negatives
• countLines vs countBlankLines
  • Low precision -> has false positives
  • Full recall -> has no false negatives
Prediction evaluation

• Note exactly fair cause the CNN and LSTM only get to see a token stream.

<table>
<thead>
<tr>
<th>Model</th>
<th>Sampled Test Set</th>
<th>Full Test Set</th>
<th>prediction rate (examples / sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Precision</td>
<td>Recall</td>
<td>F1</td>
</tr>
<tr>
<td>CNN+Attention [Allamanis et al. 2016]</td>
<td>47.3</td>
<td>29.4</td>
<td>33.9</td>
</tr>
<tr>
<td>LSTM+Attention [Iyer et al. 2016]</td>
<td>27.5</td>
<td>21.5</td>
<td>24.1</td>
</tr>
<tr>
<td>Paths+CRFs [Alon et al. 2018]</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>PathAttention (this work)</strong></td>
<td><strong>63.3</strong></td>
<td><strong>56.2</strong></td>
<td><strong>59.5</strong></td>
</tr>
</tbody>
</table>
Evaluation of Attention

• No-attention – unweighted average
• Hard-attention – only use highest ranked context
• Train-soft, predict-hard – only marginal improvement
• Element-wise soft attention – don’t compress with fully connected layer
  • Better performance but hard to interpret

Table 4. Comparison of model designs.

<table>
<thead>
<tr>
<th>Model Design</th>
<th>Precision</th>
<th>Recall</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>No-attention</td>
<td>54.4</td>
<td>45.3</td>
<td>49.4</td>
</tr>
<tr>
<td>Hard attention</td>
<td>42.1</td>
<td>35.4</td>
<td>38.5</td>
</tr>
<tr>
<td>Train-soft, predict-hard</td>
<td>52.7</td>
<td>45.9</td>
<td>49.1</td>
</tr>
<tr>
<td>Soft attention</td>
<td>63.1</td>
<td>54.4</td>
<td>58.4</td>
</tr>
<tr>
<td>Element-wise soft attention</td>
<td>63.7</td>
<td>55.4</td>
<td>59.3</td>
</tr>
</tbody>
</table>
Data Ablation Study: evaluating components

- Note performance of ‘No-value’ is significantly worse suggesting not robust with respect to changing variable names.
- The addition of “only-values” and “no-values” is less than "Full" suggesting some kind of synergy

<table>
<thead>
<tr>
<th>Path-context input</th>
<th>Precision</th>
<th>Recall</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full: $\langle x_s, p, x_t \rangle$</td>
<td>63.1</td>
<td>54.4</td>
<td>58.4</td>
</tr>
<tr>
<td>Only-values: $\langle x_s, __, x_t \rangle$</td>
<td>44.9</td>
<td>37.1</td>
<td>40.6</td>
</tr>
<tr>
<td>No-values: $\langle __, p, __ \rangle$</td>
<td>12.0</td>
<td>12.6</td>
<td>12.3</td>
</tr>
<tr>
<td>Value-path: $\langle x_s, p, __ \rangle$</td>
<td>31.5</td>
<td>30.1</td>
<td>30.7</td>
</tr>
<tr>
<td>One-value: $\langle x_s, __, __ \rangle$</td>
<td>10.6</td>
<td>10.4</td>
<td>10.7</td>
</tr>
</tbody>
</table>
Variable Names

- Not that surprising given some of the examples of methods and the absence of semantic/execution information.
Variable Names

- Not that surprising given some of the examples of methods and the absence of semantic/execution information.
- Biggest indicates dependency on names
• Not that surprising given some of the examples of methods and the absence of semantic/execution information.

• Biggest indicates dependency on names

• Completely obfuscated is basically hopeless
## Interpretation: Analogies

Table 7. Semantic analogies between method names.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>open</td>
<td>connect</td>
<td>close</td>
<td>disconnect</td>
</tr>
<tr>
<td>key</td>
<td>keys</td>
<td>value</td>
<td>values</td>
</tr>
<tr>
<td>lower</td>
<td>toLowerCase</td>
<td>upper</td>
<td>toUpperCase</td>
</tr>
<tr>
<td>down</td>
<td>onMouseDown</td>
<td>up</td>
<td>onMouseUp</td>
</tr>
<tr>
<td>warning</td>
<td>getWarningCount</td>
<td>error</td>
<td>getErrorCount</td>
</tr>
<tr>
<td>value</td>
<td>containsValue</td>
<td>key</td>
<td>containsKey</td>
</tr>
<tr>
<td>start</td>
<td>activate</td>
<td>end</td>
<td>deactivate</td>
</tr>
<tr>
<td>receive</td>
<td>download</td>
<td>send</td>
<td>upload</td>
</tr>
</tbody>
</table>
Limitations

• Closed label vocabulary:
  • Even though the labels for prediction can be composed rare names can’t be predicted.
  • Eg: `imageFormatExceptionShouldProduceNotSuccessOperationResultWithMessage`

• Sparsity
  • Variable names `newArray` and `oldArray` are treated as separate terminals.
  • AST paths that differ by a single node are considered completely different.

• Dependency on Variable Names
  • Potentially remedied by pipelining an upstream de-obfuscator tool
Questions?

• [https://code2vec.org](https://code2vec.org)
Other prediction tasks:

<table>
<thead>
<tr>
<th>Task</th>
<th>Previous works</th>
<th>AST Paths (this work)</th>
<th>Params (length/width)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Variable name prediction</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JavaScript</td>
<td>24.9% (no-paths)</td>
<td>60.0% (UnuglifyJS)</td>
<td>67.3%</td>
</tr>
<tr>
<td>Java</td>
<td>23.7% (rule-based)</td>
<td>50.1% (CRFs+4-grams)</td>
<td>58.2%</td>
</tr>
<tr>
<td>Python</td>
<td>35.2% (no-paths)</td>
<td>56.7% (top-7)</td>
<td>56.1%</td>
</tr>
<tr>
<td>C#</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Method name prediction</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JavaScript</td>
<td>44.1% (no-paths)</td>
<td></td>
<td>53.1%</td>
</tr>
<tr>
<td>Java</td>
<td>16.5%, F1: 33.9 (Allamanis et al. [7])</td>
<td>47.3%, F1: 49.9</td>
<td>6/2</td>
</tr>
<tr>
<td>Python</td>
<td>41.6% (no-paths)</td>
<td>51.1% (top-7)</td>
<td></td>
</tr>
<tr>
<td><strong>Full type prediction</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java</td>
<td>24.1% (naïve baseline)</td>
<td>69.1%</td>
<td></td>
</tr>
</tbody>
</table>

Table 2. Accuracy comparison for variable name prediction, method name prediction, and full type prediction using CRFs.
Variable Name Prediction

Stripped Names

```javascript
function f(a, b, c) {
    b.open('GET', a, false);
    b.send(c);
}
```

AST Paths + CRFs

```javascript
function f(url, request, callback) {
    request.open('GET', url, false);
    request.send(callback);
}
```

```javascript
function f(source, req, n) {
    req.open("GET", source, false);
    req.send(n);
}
```