code2vec: Learning Distributed Representations of Code

Uri Alon, Meital Zilberstein, Omer Levy, Eran Yahav

Facebook AI Research

Technion
Predicting Properties of Programs

Program \( P \) $\rightarrow$ Trained Model $\rightarrow$ Property \( \varphi \)
A Motivating Example: Semantic Labeling of Code

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

- reverseArray: 77.34%
- reverse: 18.18%
- subArray: 1.45%
Program $P$ \hspace{1cm} Trained Model \hspace{1cm} Property $\varphi$

Training data (millions of examples):

\[
(P_1, \varphi_1), \quad (P_2, \varphi_2), \quad \ldots, \quad (P_m, \varphi_m)
\]

Test data: $(P', \varphi')$
Program $P \xrightarrow{} \varphi$

Training data (millions of examples): \[
\{(P_1, \varphi_1), (P_2, \varphi_2), \ldots, (P_m, \varphi_m)\}
\]

Test data: $(P', \varphi')$
Code2vec: a neural network for predicting properties of code
Code2vec: a neural network for predicting properties of code

Example application: predicting method names

```java
String[] ______(...) {
    final String[] = ...;
    ...
    return newArray;
}
```

• A general approach – has many possible applications:
  • Yes/no malware, required dependencies, keywords / hashtags, clone detection...
How does code2vec work?
Neural Networks

- Sequences of simple algebraic functions over vectors and matrices

- **A simple example**: Predict how positive is a given sentence (regression)

\[ \text{Score} = 7.6 \quad \text{Truth} = 9.4 \]

\[ \text{Loss}(\text{pred}, \text{truth}) \]

\[ \begin{align*}
\text{vec} &\leftarrow \text{vec} - \alpha \frac{\partial \text{loss}}{\partial \text{vec}} \\
\text{w} &\leftarrow \text{w} - \alpha \frac{\partial \text{loss}}{\partial \text{w}}
\end{align*} \]
Back to our problem

Two main challenges in encoding programs:

1. How to decompose programs to smaller building blocks?
   • Small enough to repeat across programs
   • Large enough to be meaningful

2. How do we aggregate a set of these building blocks?

   the “bias-variance tradeoff”
Code2vec: High-level Overview

```
String[] ___(...) {
    final String[] = ...;
    ...
    return newArray;
}
```
Challenge #1: Decomposing Programs

Implicitly re-learn syntactic & semantic regularities

Sweet-spot

Requires expertise, language-specific, task-specific model

[“A General Path-based Representation for Predicting Program Properties”, PLDI’2018]
A Program as a Set of AST Paths

while (!done) {
    if (someCondition()) {
        done = true;
    }
}

(done, SymbolRef ↑)

- AST paths capture some of the semantics, by using only the syntax.
- We represent a program as the set of all its paths.

[“A General Path-based Representation for Predicting Program Properties”, PLDI’2018]
Representing AST-Paths as Vectors

Two sets of learned vectors:
- Token vectors
- Path vectors

\[
\text{tanh}(w \cdot [ \text{full-}\text{connected layer} ]) = \text{path\_context}
\]
**Input:** an arbitrary-sized set of vectors representing AST paths

- Select the “most important vector”

**Challenge #2: Aggregating a Set of Path-Contexts**

- Use all vectors, e.g., by averaging them
- Attention – a learned weighted average
Attention

Core idea - the values of the vectors learn two distinct goals:

1. The semantic meaning of the path-context
2. The amount of attention this path-context should get

A learned weighted average!
Predicting method names:
- **Training set**: ~14M examples
- **Training time**: <1 day (very fast) thanks to its simplicity
- **End-to-end**: the entire network is trained simultaneously
boolean f(Object target) {
    for (Object elem: this.elements) {
        if (elem.equals(target)) {
            return true;
        }
    }
    return false;
}

Object f(int target) {
    for (Object elem: this.elements) {
        if (elem.hashCode().equals(target)) {
            return elem;
        }
    }
    return this.defaultValue;
}
⇒ Attention provides interpretability!
⇒ Attention provides interpretability!
⇒ Attention provides interpretability!
The Vector Space of Target Labels

Cosine-similar vectors are learned for semantically similar labels.
The Vector Space of Target Labels

Cosine-similar vectors are learned for semantically similar labels.
http://code2vec.org

MOST SIMILAR

count

...is similar to:

<table>
<thead>
<tr>
<th>predict</th>
<th>getCount</th>
<th>70.02%</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>size</td>
<td>69.64%</td>
</tr>
<tr>
<td></td>
<td>index</td>
<td>64.99%</td>
</tr>
</tbody>
</table>
http://code2vec.org

COMBINATIONS

equals  and  toLower

...combined, are similar to:

PREDICT

equalsIgnoreCase  |  78.75%

isUpperCase  |  75.82%

equiv  |  75.72%
receive is to download as...

...is to:

| upload   | 76.38% |
| delete   | 71.53% |
| connect  | 70.51% |

http://code2vec.org
boolean f(Object target) {
    for (Object elem: this.elements) {
        if (elem.equals(target)) {
            return true;
        }
    }
    return false;
}
**Code2vec**

A neural network for learning distributed representations of code. This is an official implementation of the model described in:

Uri Alon, Marki Zilberstein, Omer Levy and Evan Yehua, "code2vec: Learning Distributed Representations of Code", POPL 2019 [PDF]
Summary

• Core ideas in learning code snippets:
  1. Representing a code snippet as a set of syntactic paths
  2. Aggregate all paths using neural attention
• A simple and fast to train architecture
• Interpretable thanks to the attention mechanism
• The learned vectors capture interesting phenomena

http://code2vec.org
How many paths do you take from each code snippet? Taking all paths is quadratic!

• An unlimited number!

• Since attention is simply a weighted average, it can handle an arbitrary number of path-contexts.

• Empirically, we found that sampling 200 from each code example is sufficient. ~200 is also the average number of paths per example.

• This number (200) can be easily increased if the dataset contained especially large pieces of code.

• Paths that are missed due to sampling are “covered” by other paths.
Why not performing additional control flow or data flow analyses?

• These might help, but we are not sure they are necessary here. Most of the important signals are expressed in the syntax.

• Our pure-syntactic approach has the advantage of generality – the same approach can be easily applied to other languages.

• Semantic analysis is probably necessary in other tasks (for example, when the programs are binaries).
How robust are the results for variable renaming?

• As any machine learning model, confusing or adversarial examples can mislead our model.

• Since the network was trained on “well-named” examples from top-starred GitHub projects, it does perform worse without names.

• We are exploring similar approaches for obfuscated code as part of ongoing research.
Do you keep vectors for all paths and tokens?

• Almost all!
  • Limiting to the most occurring 1M tokens, 1M paths, and 300k target labels.

• Each token and path vectors has 128 elements of 4 bytes (float32)
• Each target vector has 384 elements of 4 bytes
• Attention vector has 384 elements
• Fully connected layer is a matrix of size 384 × 384

• **Total size:** \( \frac{128 \cdot 4 \cdot (1M + 1M \text{ token+path vocab sizes})}{\text{vector}} + \frac{384 \cdot 4 \cdot 300k \text{ target vocab size}}{\text{vector}} + \frac{384}{\text{attention}} + \frac{384^2}{\text{fully-connected}} \approx 1.5 \text{ GB} \)

• **Standard GPU memory size:** 12 GB
Did you try Gated Graph NNs (Allamanis et al., ICLR’2018)?

• GGNNs were applied to a simpler task of Var-Misuse.
  • Their code is not fully available.

• Two conceptual advantages of code2vec over GGNNs:
  1. **Much faster to train** - thus practically easier to leverage huge corpora (our dataset is orders of magnitude larger).
  2. **Our model is purely syntactic** - the same algorithm can work for every programming language. In GGNNs, the edges in the graph are analyses like “ComputedFrom” and “LastWrite”, that need to be re-implemented for different languages.
Can a non-neural model solve the same task?

• Yes, and pretty well (PLDI’2018).
  • But not as good as a neural model.

• Main advantages of using a neural network:
  1. Much **better generalization** (Section 5 in the paper)
  2. Our neural network can **produce a vector**, which can be fed to a variety of other (neural and non-neural) ML models and tasks.