A General Path-Based Representation for Predicting Program Properties

Uri Alon, Meital Zilberstein, Omer Levy, Eran Yahav

University of Washington
Motivating Example #1
Prediction of Variable Names in Python

def sh3(c):
    p = Popen(c, stdout=PIPE, stderr=PIPE, shell=True)
    o, e = p.communicate()
    r = p.returncode
    if r:
        raise CalledProcessError(r, c)
    else:
        return o.rstrip(), e.rstrip()

def sh3(cmd):
    process = Popen(cmd, stdout=PIPE, stderr=PIPE, shell=True)
    out, err = process.communicate()
    retcode = process.returncode
    if retcode:
        raise CalledProcessError(retcode, cmd)
    else:
        return out.rstrip(), err.rstrip()
Motivating Example #2
Prediction of Method Names in JavaScript

```javascript
function cloneObject(object) {
    if (!object)
        return object;
    var clone = {};
    for (var key in object) {
        clone[key] = object[key];
    }
    return clone;
}
```
Motivating Example #3
Prediction of full types in Java

StackOverflow answer:

```java
Configuration conf = HBaseConfiguration.create();
try {
    Connection connection = ConnectionFactory.createConnection(conf);
}

import org.apache.hadoop.hbase.client.Connection;
com.mysql.jdbc.Connection ??
org.apache.http.Connection ??
import org.apache.hadoop.hadoop.hbase.client.Connection;
```
Previously – separate techniques for each problem / language

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>JavaScript</th>
<th>Python</th>
<th>C#</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Variable name prediction</strong></td>
<td>Bichsel et al. CCS’2016 (CRFs) ✔️</td>
<td>Raychev et al. POPL’2015 (CRFs) ✔️</td>
<td>Raychev et al. OOPSLA’2016 (Decision Trees) ✔️</td>
<td>...</td>
<td>✔️</td>
</tr>
<tr>
<td><strong>Method name prediction</strong></td>
<td>Allamanis et al. ICML’2016 (NNS) ✔️</td>
<td>Raychev et al. OOPSLA’2016 (Decision Trees) ✔️</td>
<td></td>
<td>...</td>
<td>✔️</td>
</tr>
<tr>
<td><strong>Full types prediction</strong></td>
<td></td>
<td></td>
<td></td>
<td>Allamanis et al. ICML’2015 (Generative) ✔️</td>
<td>...</td>
</tr>
</tbody>
</table>

*Completely automatically!*
How to represent a program element?

- Should work for many programming languages
- Should work for different tasks
- Useful in multiple learning algorithms

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

```java
while (!count) {
    if (someCondition()) {
        count = true;
    }
}
```

- What are the properties that make “done” a “done”?
How to represent a program element?

Key idea:

The semantic role of a program element is the set of all **structured contexts** in which it appears.

“done” is “done” because it appears in particular structured contexts.
A general and simple method to represent code in machine learning models is represented as the set of all its paths.

For example:

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

`done` is represented as the set of all its paths.
while (!done) {
    if (someCondition()) {
        done = true;
    }
}

while (!x) {
    foo();
    if (bar() < 3) {
        log.info(zoo);
        x = true;
    }
}
Advantages of AST-Paths representation

✓ Expressive enough to capture any property that is expressed syntactically.
✓ Independent of the programming language
✓ Automatically extractable – only requires a parser
✓ Not bound to the learning algorithm
✓ Works for different tasks
Predicting program properties with AST paths

- Off-the-shelf algorithms
- Plug-in our representation

Conditional Random Fields (CRFs)  

word2vec-based
Predicting properties with CRFs

- **Nodes**: program elements
- **Factors**: learned scoring functions:
  - \((Values, Values, Paths) \rightarrow \mathbb{R}\)
- The same as in (JSNice, Raychev et al., POPL’2015), but with our paths as factors
Predicting properties with word2vec

- **Input**: pairs of: \((word, context)\)

- **Model**:
  - word vectors: \(W_{vocab}\)
  - context vectors: \(C_{vocab}\)

- **Prediction**:
  - \(\text{predict}(\{\overrightarrow{c_1}, \ldots, \overrightarrow{c_n}\}) = \arg\max_{w_i \in W_{vocab}} [w_i \cdot \sum_j c_j]\)
Word2vec and different contexts

- Input: pairs of: \((word, context)\)
- Train word2vec with 3 types of contexts:
  - Neighbor tokens
  - Surrounding AST-nodes
  - AST paths
Word2vec and different contexts

- Input: pairs of: $(word, context)$
- Train word2vec with 3 types of contexts:
  - Neighbor tokens
  - Surrounding AST-nodes
  - AST paths
Word2vec and different contexts

- Input: pairs of: \((word, context)\)
- Train word2vec with 3 types of contexts:
  - Neighbor tokens
  - Surrounding AST-nodes
  - AST paths
Word2vec and different contexts

- Input: pairs of: \((\text{word}, \text{context})\)
- Train word2vec with 3 types of contexts:
  - Neighbor tokens
  - Surrounding AST-nodes
  - AST paths
Evaluation

- 4 programming languages
  - Java, JavaScript, Python, C#
- 3 tasks
  - predicting method names, variable names, full types ("...hbase.client.Connection")
- 2 learning algorithms
  - CRFs, word2vec-based
Predicting variable names with CRFs

Format: Absolute (Relative%)

<table>
<thead>
<tr>
<th></th>
<th>Accuracy (%)</th>
<th>CRFs + n-grams</th>
<th>UnuglifyJS</th>
<th>CRFs + No-relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>+8.1 (16.2%)</td>
<td>+7.3 (12.2%)</td>
<td>+21.5 (61%)</td>
<td></td>
</tr>
<tr>
<td>JavaScript</td>
<td>60</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Python</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C#</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Word2vec with different context types

Task: Variable names, word2vec, JavaScript
▪ Limiting path-length and path-width
  ▪ Path vocabulary size (JavaScript):

  length: 7 → 6: $13M \rightarrow 11M$
  width: 3 → 2: $13M \rightarrow 12M$

▪ Path abstraction
  ▪ Path vocabulary size (Java):

  $\sim 10^7 \rightarrow \sim 10^2$
Effect of limiting path length and width

Task: Variable names, CRFs, JavaScript

Accuracy (%) vs. Max path-length

- AST Paths with max_width=3
- AST Paths with max_width=2
- AST Paths with max_width=1
- UnuglifyJS
AST Path Abstractions

**Task:** Variable names, CRFs, Java

![Graph showing the relationship between Accuracy (%) and Training time (hours)]

- **no-path**
- **first-last**
- **top**
- **first-top-last**
- **forget-order**
- **no-arrows**
- **full**

SymbolRef ↑ UnaryPrefix! ↑ While ↓ If ↓ Assign= ↓ SymbolRef

SymbolRef ↑ While ↓ SymbolRef

only values, without considering the relation between them
Example (JavaScript)

```javascript
function countSomething(x, t) {
    var c = 0;
    for (var i = 0, l = x.length; i < l; i++) {
        if (x[i] === t) {
            c++;
        }
    }
    return c;
}
```
Example (JavaScript)

```javascript
function countSomething(array, target) {
    var count = 0;
    for (var i = 0, l = array.length; i < l; i++) {
        if (array[i] === target) {
            count++
        }
    }
    return count;
}
```
Example (Java)

```java
public String sendGetRequest(String l) {
    HttpClient c = HttpClientBuilder.create().build();
    HttpGet r = new HttpGet(l);
    String u = USER_AGENT;
    r.addHeader("User-Agent", u);
    HttpResponse s = c.execute(r);
    HttpEntity t = s.getEntity();
    String g = EntityUtils.toString(t, "UTF-8");
    return g;
}
```
public String sendGetRequest(String url) {
    HttpClient client = HttpClientBuilder.create().build();
    HttpGet request = new HttpGet(url);
    String user = USER_AGENT;
    request.addHeader("User-Agent", user);
    HttpResponse response = client.execute(request);
    HttpEntity entity = response.getEntity();
    String result = EntityUtils.toString(entity, "UTF-8");
    return result;
}
# Semantic Similarity Between Names

## CRFs

```javascript
var d = false;
while (!d) {
    doSomething();
    if (someCondition()) {
        d = true;
    }
}
```

<table>
<thead>
<tr>
<th>Candidate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
</tr>
<tr>
<td>2.</td>
</tr>
<tr>
<td>3.</td>
</tr>
<tr>
<td>4.</td>
</tr>
<tr>
<td>5.</td>
</tr>
<tr>
<td>6.</td>
</tr>
<tr>
<td>7.</td>
</tr>
<tr>
<td>8.</td>
</tr>
</tbody>
</table>
More Semantic Similarities

<table>
<thead>
<tr>
<th>Similarities</th>
</tr>
</thead>
<tbody>
<tr>
<td>req ~ request</td>
</tr>
<tr>
<td>count ~ counter ~ total</td>
</tr>
<tr>
<td>element ~ elem ~ el</td>
</tr>
<tr>
<td>array ~ arr ~ ary ~ list</td>
</tr>
<tr>
<td>res ~ result ~ ret</td>
</tr>
<tr>
<td>i ~ j ~ index</td>
</tr>
</tbody>
</table>
Summary: a trade-off between learning effort and generalizability

- Surface text – too noisy
- Complex analyses are great, but specific to language and task
- AST paths – sweet spot of simplicity, expressivity and generalizability
- “Structural n-grams”
- A strong baseline for any machine learning for code task

Questions?