Analyzing Software using Deep Learning

RNN-based Code Completion and Repair (Part 2)

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Summer 2020

Overview

- **Recurrent neural networks (RNNs)**
- **Code completion with statistical language models**

  Based on PLDI 2014 paper by Raychev et al.
- **Repair of syntax errors**

  Based on ”Automated correction for syntax errors in programming assignments using recurrent neural networks” by Bhatia & Singh, 2016

Code Completion

- Given: Partial program with one or more holes
- Goal: Find suitable code to fill into the holes
- Basic variants in most IDEs
- Here: Fill holes with sequences of method calls
  - Which methods to call
  - Which arguments to pass

Example

```
SmsManager smsMgr = SmsManager.getDefault();
int length = message.length();
if (length > MAX_SMS_MESSAGE_LENGTH) {
  ArrayList<String> msgList =
      smsMgr.divideMsg(message);
  // hole H1
} else {
  // hole H2
}
```

Statistical Language Model

- Dictionary of words
- Sentences: sequences of words
- Model: Probability distrib. over all possible sentences

Example: English

$Pr(\text{"hello world"}) > Pr(\text{"world hello"})$

- Most basic model: Predict next word based on all previous words

$$Pr(s) = \prod_{i=1}^{m} Pr(w_i \mid h_{i-1}) \qquad \text{where } s = w_1 \cdot \ldots \cdot w_m$$

$$h_i = w_1 \cdot \ldots \cdot w_i$$

Model-based Code Completion

- **Program code $\approx$ sentences in a language**
- **Code completion $\approx$ Finding the most likely completion of the current sentence**

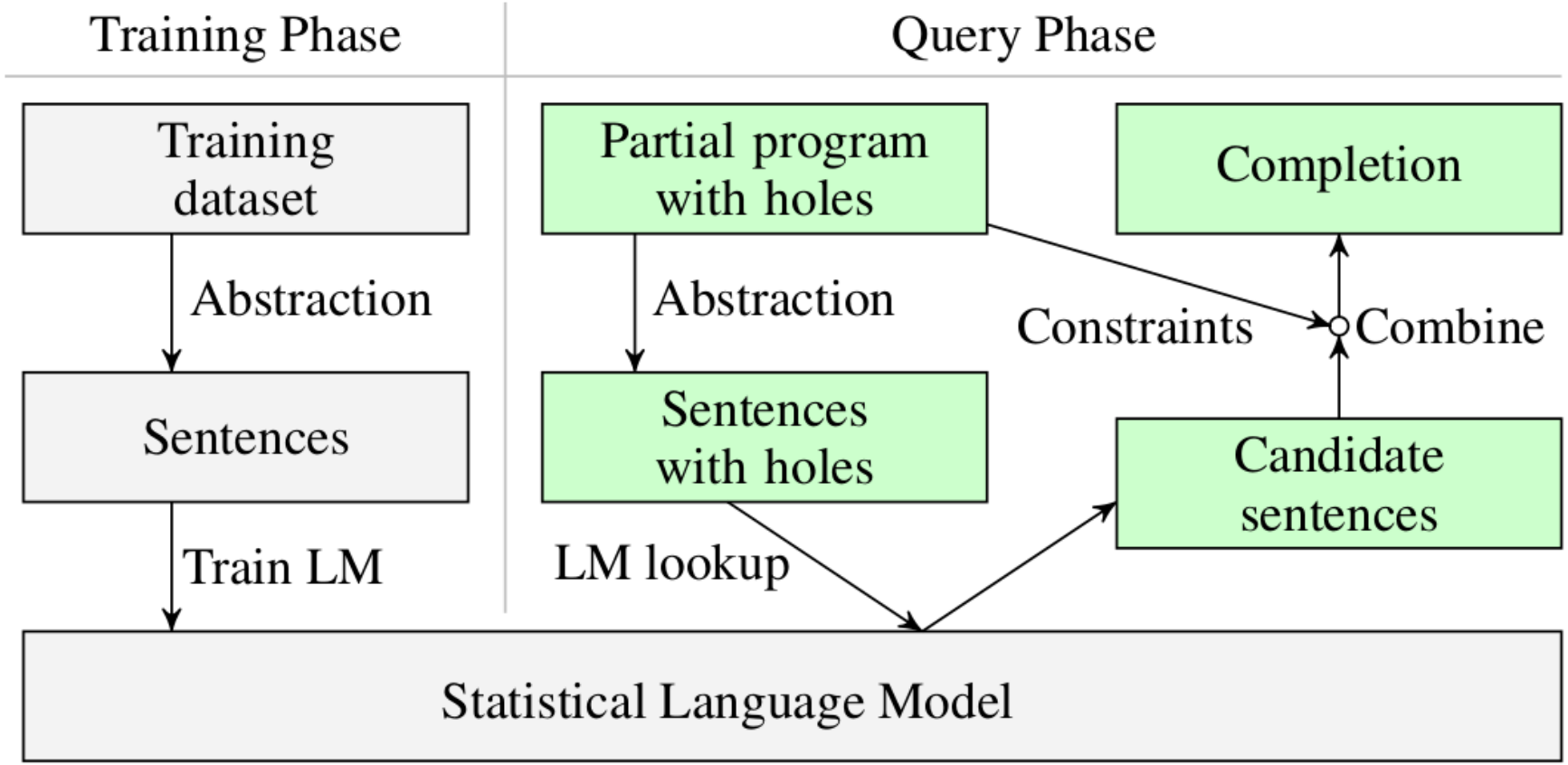
Model-based Code Completion

- **Program code $\approx$ sentences in a language**
- **Code completion $\approx$ Finding the most likely completion of the current sentence**

Challenges

- How to abstract code into sentences?
- What kind of language model to use?
- How to efficiently predict a completion

Overview of SLANG Approach



From ”Code Completion with Statistical Language Models”
by Raychev et al., 2014

n-gram Language Model

Pb. with "all history" model: Training data may not contain anything about h_i

Idea: Next word depends on $n-1$ previous words

$$Pr(s) = \prod_{i=1}^{m} Pr(w_i \mid w_{i-(n-1)} \cdot \ldots \cdot w_{i-1})$$

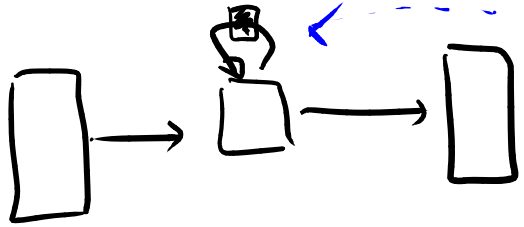
Example: $Pr(\text{to} \cdot \text{be} \cdot \text{or} \cdot \text{not} \cdot \text{to} \cdot \text{be})$ $\qquad n = 3$

$= Pr(\text{to} \mid \cdot\varepsilon) \cdot Pr(\text{be} \mid \text{to}) \cdot Pr(\text{or} \mid \text{to} \cdot \text{be})$

$\cdot \ldots \cdot Pr(\text{be} \mid \text{not} \cdot \text{to})$

Probab. of n-grams: Estimated from corpus of training examples

RNN-based model



store information about (all) previous words

previous word

next word

Encoded as vector: One-hot encoding

Length = size of vocabulary

All values are zero, except position of specific word set to one

0	0	0	0	...	0	1	0	0	0	0	0	0

Sequences of Method Calls

Abstracting code into sentences

- Method call $\approx$ word
- Sequence of method calls $\approx$ sentence
- Separate sequences for each object
- Objects can occur in call as base object, argument, or return value

Option 1: Dynamic Analysis

Execute program and observe each method call

Advantage:

- Precise results

Disadvantage:

- Only analyzes executed code

```
if (getInput() > 5) {  // Suppose always taken
  obj.foo();           // in analyzed execution
} else {
  obj.bar();  // Never gets analyzed
}
```

Option 2: Static Analysis

Reason about execution without executing the code

Advantage:

- Can consider all execution paths

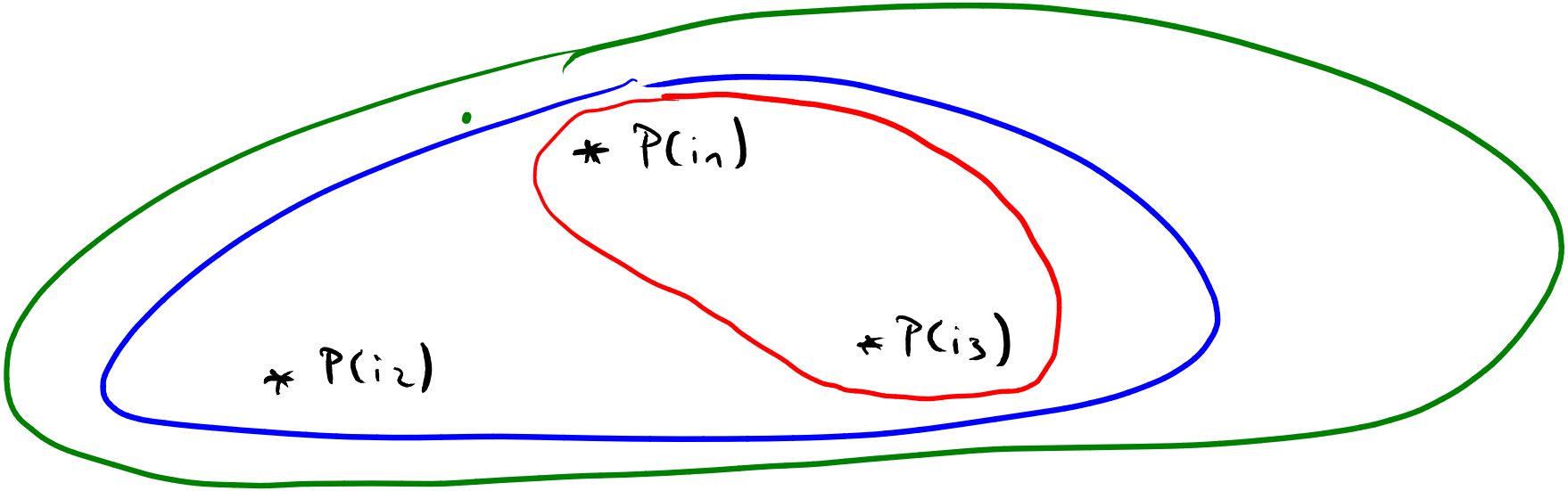
Disadvantage:

- Need to abstract and approximate actual execution

```
if (getInput() > 5) {
  a.foo();  // Does this call ever get executed?
}
b.bar();    // May a and b point to the same object?
```

Over- & Underapproximation

Program P, Input i, Behavior P(i)



All possible behaviors (what we want to analyze, ideally)

Underapproximation (most dynamic analyses)

Overapproximation (most static analyses)

Static Analysis of Call Sequences

SLANG approach: Static analysis

- Bound the number of analyzed loop iterations
- On control flow joins, take union of possible execution sequences
- Points-to analysis to reason about references to objects

Example

```
SmsManager smsMgr = SmsManager.getDefault();
int length = message.length();
if (length > MAX_SMS_MESSAGE_LENGTH) {
  ArrayList<String> msgList =
      smsMgr.divideMsg(message);
} else {}
```

Example

```
SmsManager smsMgr = SmsManager.getDefault();
int length = message.length();
if (length > MAX_SMS_MESSAGE_LENGTH) {
  ArrayList<String> msgList =
      smsMgr.divideMsg(message);
} else {}
```

5 sequences:

Object	Calls
smsMgr	(getDefault, ret)
smsMgr	(getDefault, ret) · (divideMsg, 0)
message	(length, 0)
message	(length, 0) · (divideMsg, 1)
msgList	(divideMsg, ret)

Training Phase

- Training data used for paper:
  3 million methods from various Android projects
- Extract sentences via static analysis
- Train statistical language model
  - Both n-gram and RNN model

Query Phase

- Given: Method with holes
- For each hole:
  - Consider all possible completions of the partial call sequence
  - Query language model to obtain probability
    - Average of n-gram and RNN models
- Return completed code that maximizes overall probability

Example

```
SmsManager smsMgr = SmsManager.getDefault();
int length = message.length();
if (length > MAX_SMS_MESSAGE_LENGTH) {
  ArrayList<String> msgList =
      smsMgr.divideMsg(message);
  // hole H1
} else {
  // hole H2
}
```

Example

```
SmsManager smsMgr = SmsManager.getDefault();
int length = message.length();
if (length > MAX_SMS_MESSAGE_LENGTH) {
  ArrayList<String> msgList =
      smsMgr.divideMsg(message);
  smsMgr.sendMultipartTextMessage(..., msgList, ...);
} else {
  smsMgr.sendTextMessage(..., message, ...);
}
```

Scalability Tricks

Search space of possible completions:
Too large to explore in reasonable time

Refinements to reduce space

- Users may provide hints
  - How many calls to insert
  - Which objects to use
- Replace infrequent words with ”unknown”
- Obtain candidate calls using bi-gram model
- Query language model only for candidates