Clone Detection and Maintenance with AI Techniques

Na Meng
Virginia Tech
Code Clones

Developers copy and paste code to improve programming productivity

Clone detections tools are needed to help bug fixes or refactor code
Existing Clone Detection Tools

Code Clone Detection

- Text-based
- Token-based
- Tree-based
- Metrics-based
- Graph-based
Problem Statement

• Each algorithm works better for certain kinds of clones, but worse for the others – E.g., token-based, tree-based

• The algorithms do not prioritize clones based on their likelihoods of being refactored
Research Questions

How can we automatically characterize the similarity between clones?

How can we only report clones that are likely to be refactored by developers?
CCLEARNER: A DEEP LEARNING-BASED CLONE DETECTION APPROACH
Methodology

Code Clone Detection Problem

Classification Problem
Our Hypothesis

- Code clones are more likely to share certain kinds of tokens than other tokens

<table>
<thead>
<tr>
<th>Tokens likely to be shared</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keywords, method names, ...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tokens less likely to be shared</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable names, literals, ...</td>
</tr>
</tbody>
</table>
**Feature Extraction**

- \( \text{method}_A \rightarrow \text{method}_B \)
- \( \text{token freq list}_A \rightarrow [\text{token freq cat}_{A1}, \ldots, \text{token freq cat}_{A8}] \)

<table>
<thead>
<tr>
<th>Category Name</th>
<th>Example</th>
<th>Category Name</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved words</td>
<td>&lt;if, 2&gt;</td>
<td>Type identifiers</td>
<td>&lt;URLConnection, 1&gt;</td>
</tr>
<tr>
<td>Operators</td>
<td>&lt;+=, 2&gt;</td>
<td>Method identifiers</td>
<td>&lt;openConnection, 1&gt;</td>
</tr>
<tr>
<td>Markers</td>
<td>&lt;;,, 2&gt;</td>
<td>Qualified names</td>
<td>&lt;arr.length, 1&gt;</td>
</tr>
<tr>
<td>Literals</td>
<td>&lt;1.3, 2&gt;</td>
<td>Variable identifiers</td>
<td>&lt;conn, 2&gt;</td>
</tr>
</tbody>
</table>
Feature Extraction

\[ \text{method}_A \downarrow \text{method}_B \]

\[ \text{token}_\text{freq}_\text{list}_A \downarrow \text{token}_\text{freq}_\text{list}_B \]

\[ [\text{token}_\text{freq}_\text{cat}_{A1}, \ldots, \text{token}_\text{freq}_\text{cat}_{A8}] \]

\[ [\text{token}_\text{freq}_\text{cat}_{B1}, \ldots, \text{token}_\text{freq}_\text{cat}_{B8}] \]

\[ [\text{sim}_\text{score}_1, \ldots, \text{sim}_\text{score}_{8}] \]
Training

• Input
  – Clones
    \([sim\_score_1, \ldots, sim\_score_8], 1\]  
  – Non-clones
    \([sim\_score_1, \ldots, sim\_score_8], 0\]

• Training Process
  – DeepLearning4j*

• Output
  – A well-trained classifier (.mdl)

Testing

• Input
  – A codebase

• Output
  – 2 nodes in DNN
  – Predict the likelihood of clones and non-clones

• Challenges
  – Time cost $O(n^2)$

• Solution
  – Two filters
Evaluation

• Benchmark: BigCloneBench*
  – 10 source code folders
  – One database of ground truth
  – Clone Type: T1, T2, VST3, ST3, MT3 and WT3/4

• Data Set Construction
  – Training Data (Folder #4)
    • T1, T2, VST3 and ST3 clones
    • Randomly choose a subset of false clone pairs
  – Testing data (Other 9 folders)
    • All source files

Evaluation

• Recall

\[ R_{T1-ST3} = \frac{\text{# of retrieved true clones pairs of T1-ST3}}{\text{# of known true clones pairs of T1-ST3}} \]

• Precision

\[ P_{\text{estimated}} = \frac{\text{# of retrieved true clones pairs}}{385 \text{ detected clone pair samples}} \]

• F score

\[ F_{T1-ST3} = \frac{2 \times P_{\text{estimated}} \times R_{T1-ST3}}{P_{\text{estimated}} + R_{T1-ST3}} \]
## Evaluation Results

<table>
<thead>
<tr>
<th>Recall (%)</th>
<th>CCLearner</th>
<th>SourcererCC</th>
<th>NiCad</th>
<th>Deckard</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>96</td>
</tr>
<tr>
<td>T2</td>
<td>98</td>
<td>97</td>
<td>85</td>
<td>82</td>
</tr>
<tr>
<td>VST3</td>
<td>98</td>
<td>92</td>
<td>98</td>
<td>78</td>
</tr>
<tr>
<td>ST3</td>
<td>89</td>
<td>67</td>
<td>77</td>
<td>78</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Precision (%)</th>
<th>CCLearner</th>
<th>SourcererCC</th>
<th>NiCad</th>
<th>Deckard</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>93</td>
<td>98</td>
<td>68</td>
<td>71</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>F (%)</th>
<th>CCLearner</th>
<th>SourcererCC</th>
<th>NiCad</th>
<th>Deckard</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>93</td>
<td>88</td>
<td>76</td>
<td>77</td>
</tr>
</tbody>
</table>
Things We Learnt

- **CCLearner** achieves a better trade-off between precision and recall
  - Perhaps deep learning or our unique feature sets play the role
- **CCLearner’s recall goes down as more variation exists between clone peers**
  - We may need more features to represent the semantic equivalence between clones instead of purely the syntactic similarity
CLONERECOMMENDER: MACHINE LEARNING-BASED CLONE RECOMMENDATION FOR REFACTORING
Motivation

• With clone detection, developers apply clone removal refactorings
  – e.g., Extract Method and Form Template Method
• Each clone detection tool reports too many clones
• Some clones are more likely to be refactored than others
  – E.g., repetitively updated code clones vs. inactive code clones
Our Hypotheses

• The clones refactored by developers and those not refactored should manifest certain differences
  – Content, context, and evolution of each clone
  – The textual similarity/difference and co-change relationships between clone peers

• Different developers make refactoring decisions in similar or predictable ways
Approach

Feature Extraction

Per-Clone Features (15)
- Code Feature (9)
- History Features (6)

Clone Group Features (16)
- Location/Context Features (4)
- Syntactic Difference Features (6)
- Co-Change Features (5)
- Group Size (1)

Machine Learning

Testing

Classifier

Training

Clones to Refactor

Clones not to Refactor

A Project Repository

Clone Detection

Detected Clones

Detected

Non-Refactored

Refactored Clones

Clones

Non-Refactored

Clones
Things We Learnt

• The refactoring decisions seem to be predictable in most cases
• Some refactorings seem to be unpredictable
  – We can only predict refactorings based on code history and its current version
  – Some developers make different refactoring decisions
Conclusion

• Our investigation on CCLearner and CloneRecommender demonstrates that AI can improve the efficiency of software development and reduce maintenance cost

• AI techniques cannot fully overtake coding tasks due to (1) the difficulty of reasoning semantics, and (2) lack of the domain knowledge to evolve software