Clone Detection and Maintenance with AI Techniques

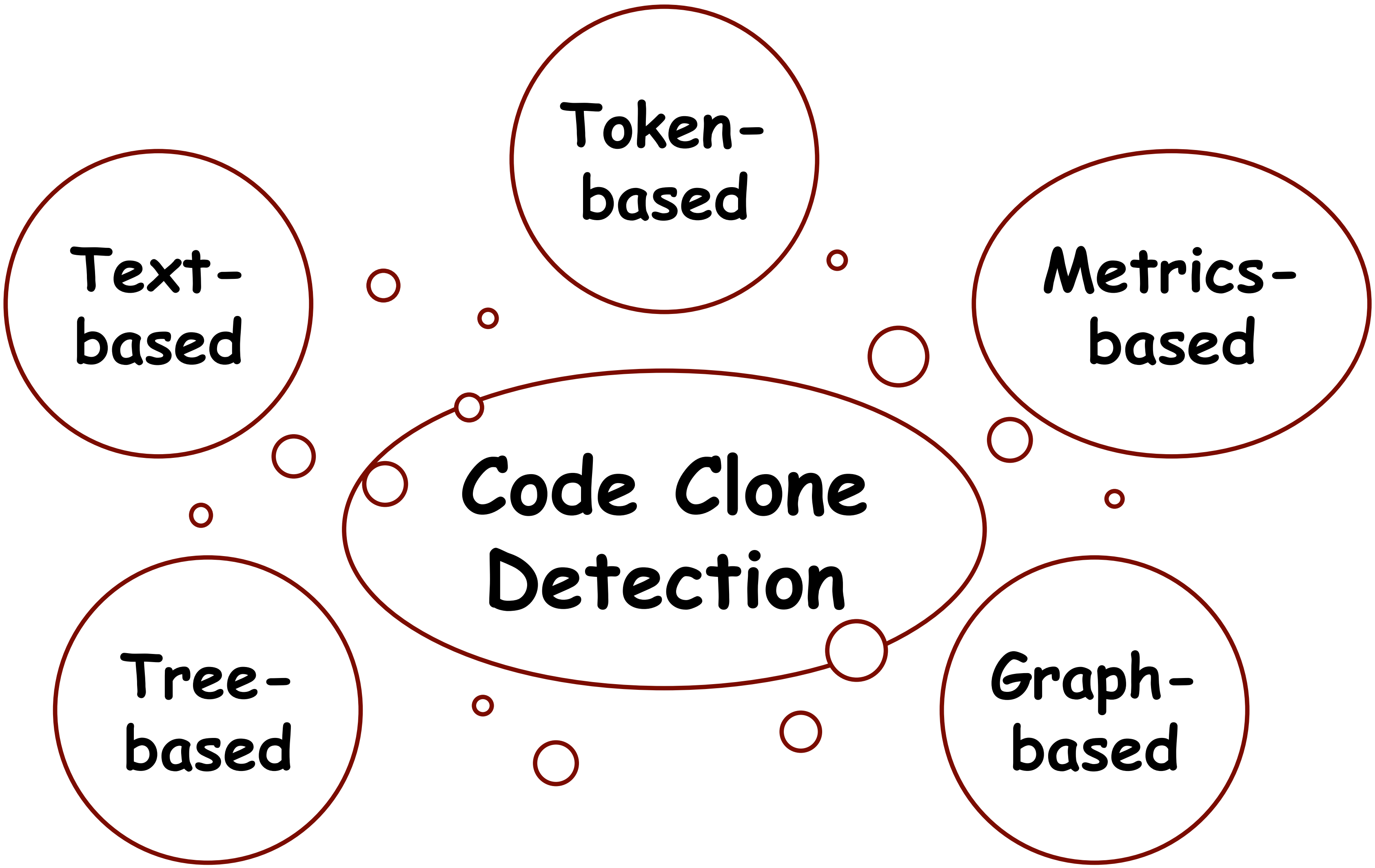
Na Meng
Virginia Tech

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



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

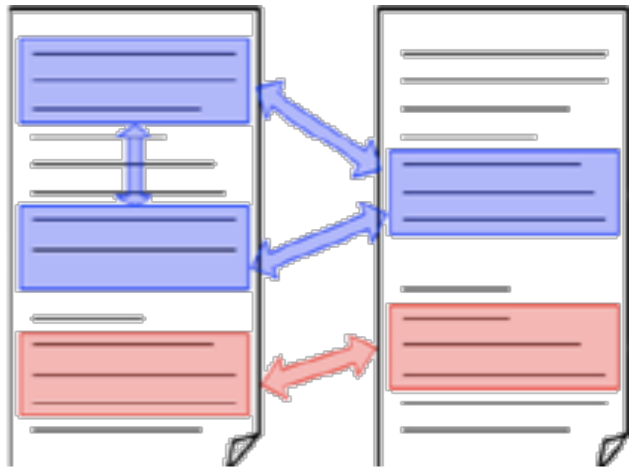
How can we automatically characterize the similarity between clones?

How can we only report clones that are likely to be refactored by developers?

[ICSME '17]: Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, Barbara Ryder

CCLEARNER: A DEEP LEARNING-BASED CLONE DETECTION APPROACH

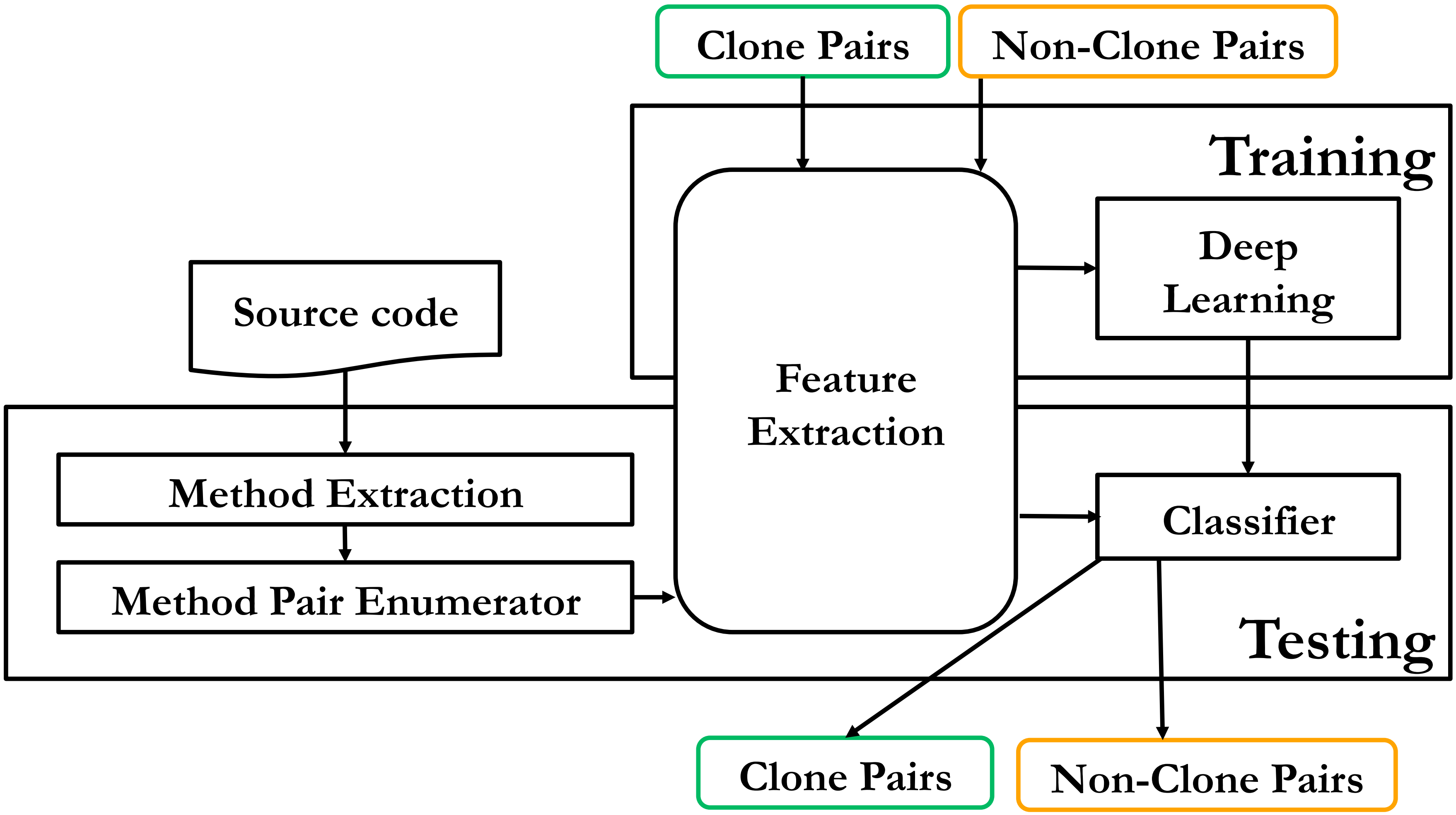
Methodology



Code Clone Detection Problem

Classification Problem

Approach


Clone Pairs
Non-Clone Pairs
Training
Deep
Learning
Source code
Feature
Extraction
Method Extraction
Method Pair Enumerator
Classifier
Testing
Clone Pairs
Non-Clone Pairs

Our Hypothesis

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

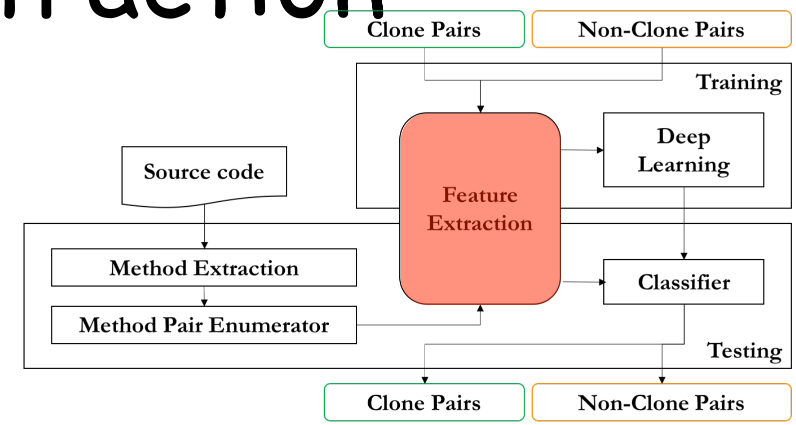
Tokens likely to be shared

Keywords, method names, …

Tokens less likely to be shared

Variable names, literals, …

Feature Extraction



$method_A$ $\quad$ $method_B$

$\downarrow$

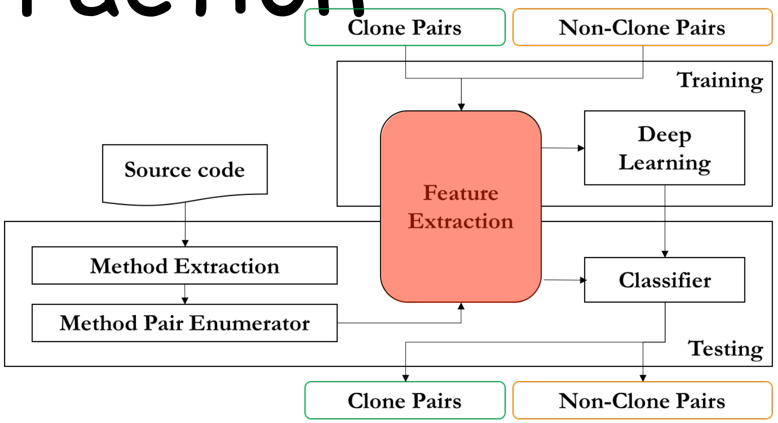
$token\_freq\_list_A$

$\downarrow$

$[token\_freq\_cat_{A1}, \ldots, token\_freq\_cat_{A8}]$

Category Name	Example	Category Name	Example
Reserved words	<if, 2>	Type identifiers	<URLConnection, 1>
Operators	<+=, 2>	Method identifiers	<openConnection, 1>
Markers	<;, 2>	Qualified names	<arr.length, 1>
Literals	<1.3, 2>	Variable identifiers	<conn, 2>

Feature Extraction



$method_A$ $\downarrow$ $token\_freq\_list_A$ $\downarrow$

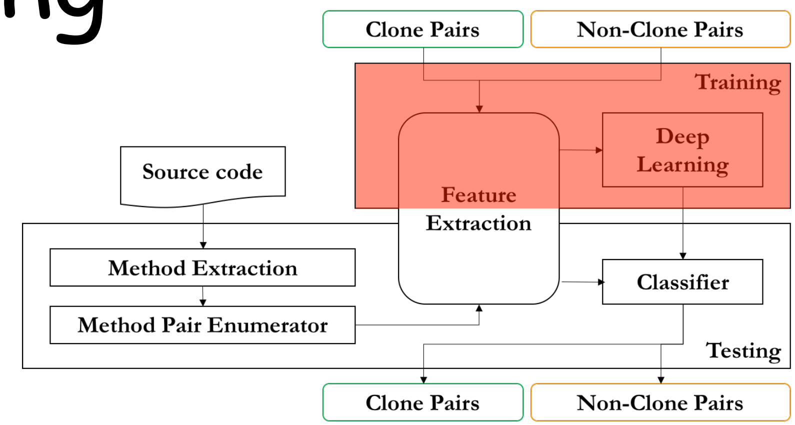
$method_B$ $\downarrow$ $token\_freq\_list_B$

$[token\_freq\_cat_{A1}, \ldots, token\_freq\_cat_{A8}]$

$[token\_freq\_cat_{B1}, \ldots, token\_freq\_cat_{B8}]$

$[sim\_score_1, \ldots, sim\_score_8]$

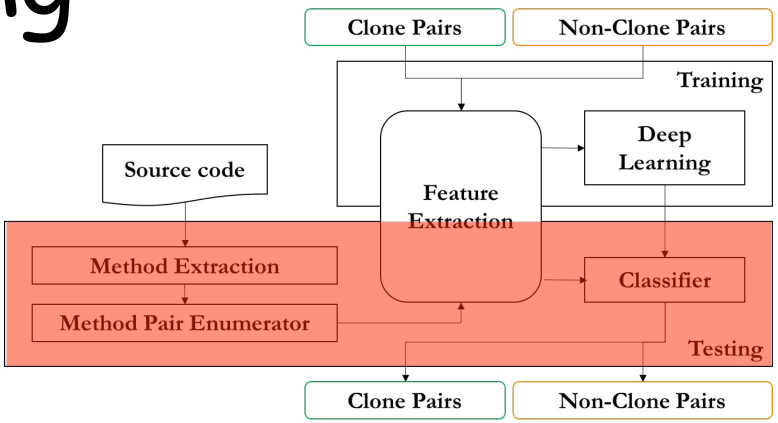
Training



- 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)

* "Deeplearning4j," http://deeplearning4j.org/, accessed: 2017-09-04

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

* Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy and Mohammad Mamun Mia, "Towards a Big Data Curated Benchmark of Inter-Project Code Clones", In Proceedings of the Early Research Achievements track of the 30th International Conference on Software Maintenance and Evolution (ICSME 2014), 5 pp., Victoria, Canada, September 2014.

Evaluation

- Recall

$$R_{T1-ST3} = \frac{\textit{\# of retrieved true clones pairs of T1-ST3}}{\textit{\# of known true clones pairs of T1-ST3}}$$

- Precision

$$P_{estimated} = \frac{\textit{\# of retrieved true clones pairs}}{\textit{385 detected clone pair samples}}$$

- F score

$$F_{T1-ST3} = \frac{2 * P_{estimated} * R_{T1\text{-}ST3}}{P_{estimated} + R_{T1\text{-}ST3}}$$

Evaluation Results

Recall (%)

	CCLearner	SourcererCC	NiCad	Deckard
T1	100	100	100	96
T2	98	97	85	82
VST3	98	92	98	78
ST3	89	67	77	78

Precision (%)

CCLearner	SourcererCC	NiCad	Deckard
93	98	68	71

F (%)

CCLearner	SourcererCC	NiCad	Deckard
93	88	76	77

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

[Under submission] Ruru Yue, Zhe Gao, Na Meng, Yingfei Xiong, Xiaoyin Wang

CLONERECOMMENDER: MACHINE LEARNING-BASED CLONE RECOMMENDATION FOR REFACTORING

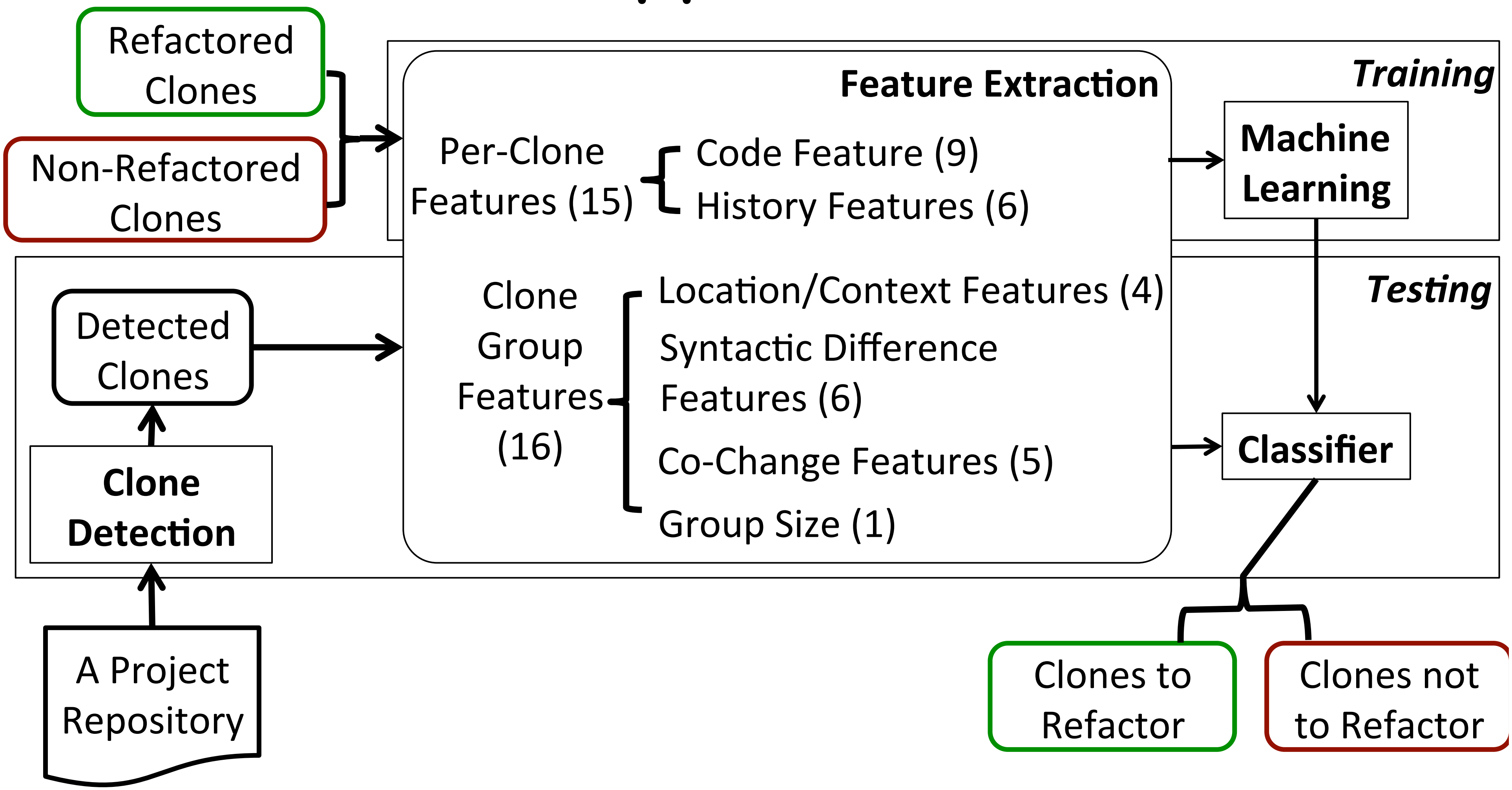
Motivation

- 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



Refactored Clones
Non-Refactored Clones
Feature Extraction
Training
Per-Clone Features (15)
Code Feature (9)
History Features (6)
Machine Learning
Testing
Detected Clones
Clone Group Features (16)
Location/Context Features (4)
Syntactic Difference Features (6)
Co-Change Features (5)
Group Size (1)
Classifier
Clone Detection
A Project Repository
Clones to Refactor
Clones not to Refactor

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