Predicting Mutation Score Using Source Code and Test Suite Metrics

RAISE 2012

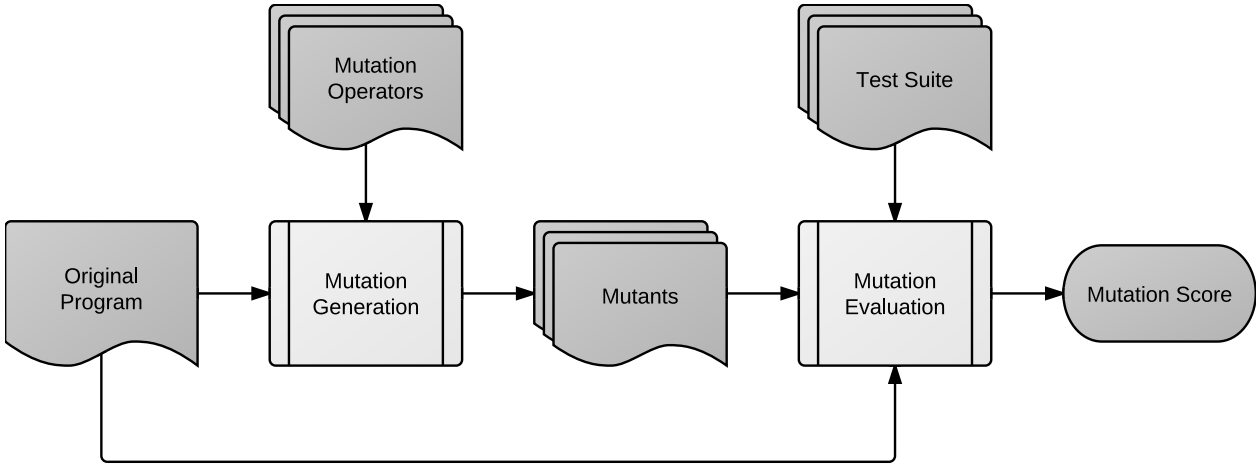
Kevin Jalbert and Jeremy S. Bradbury

{kevin.jalbert,jeremy.bradbury}@uoit.ca

Software Quality Research Group (sqrg.ca)
University of Ontario Institute of Technology
Oshawa, Ontario, Canada

June 5^{th}, 2012

Mutation Testing



- Create mutants from original program using mutation operators
- Compare mutant's results against original program using test suite
- Mutation score is percent of non-equivalent mutants killed

Mutation Operators

Name	Description
REPLACE_CONSTANT	Replace a constant
NEGATE_JUMP	Negate jump condition
ARITHMETIC_REPLACE	Replace arithmetic operator
REMOVE_CALL	Remove method call
REPLACE_VARIABLE	Replace variable reference
ABSOLUTE_VALUE	Insert absolute value of a variable
UNARY_OPERATOR	Insert unary operator

Mutation Operators

Name	Description
REPLACE_CONSTANT	Replace a constant
NEGATE_JUMP	Negate jump condition
ARITHMETIC_REPLACE	Replace arithmetic operator
REMOVE_CALL	Remove method call
REPLACE_VARIABLE	Replace variable reference
ABSOLUTE_VALUE	Insert absolute value of a variable
UNARY_OPERATOR	Insert unary operator

Example Operator – ARITHMETIC_REPLACE

Correct Program

```
class Counter {
  Integer current = new Integer(1);
  Integer limit = new Integer(10);
  public Integer add() throws Exception {
    if (this.current > this.limit) {
      throw new Exception();
    }
    return ++this.current;
  }
}
```

Mutant Program

```
class Counter {
  Integer current = new Integer(1);
  Integer limit = new Integer(10);
  public Integer add() throws Exception {
    if (this.current > this.limit) {
      throw new Exception();
    }
    return --this.current;
  }
}
```

Problem

- Mutation testing is an effective yet costly coverage technique
  - A fault-based coverage technique
  - Closest measure to test suite effectiveness
- Mutation testing would have to be applied frequently during development

Our Solution

"Our proposed approach uses machine learning to predict the mutation score based on a combination of source code and test suite metrics of the code unit under test"

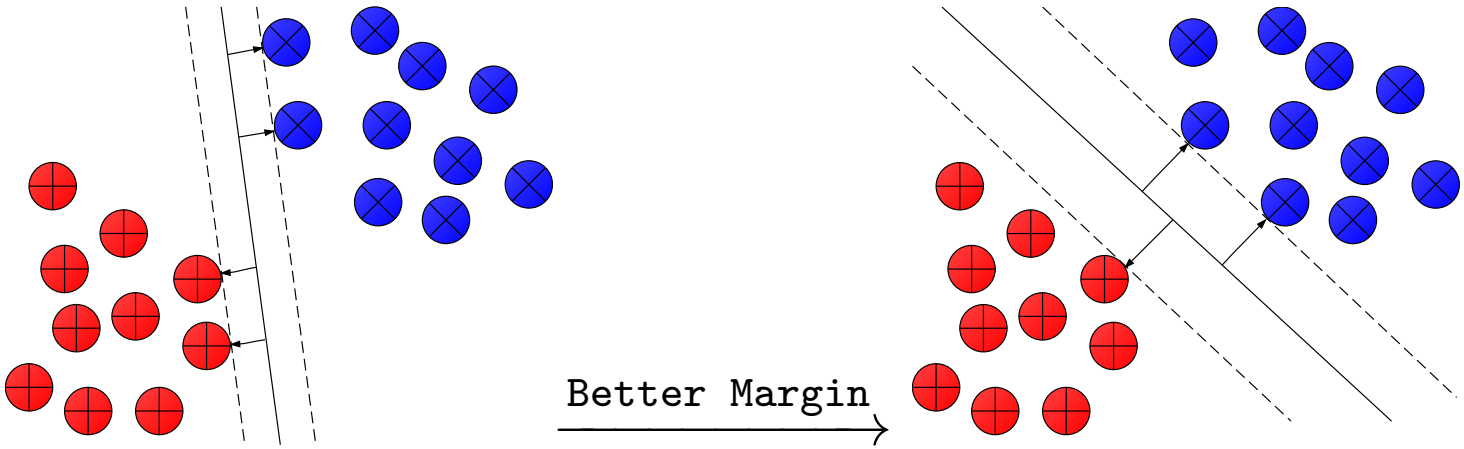
- A "do fewer and smarter" technique for mutation testing
  - Identify source code units that have low/high coverage
  - Ability to prioritize mutation testing for specific mutants

Our Approach

- Collect feature data for source code units
  - Source code metrics
  - Test suite metrics
- Collect category data for source code units
  - Mutation score for source code units
- Train and predict class and method source code units using a Support Vector Machine (SVM)

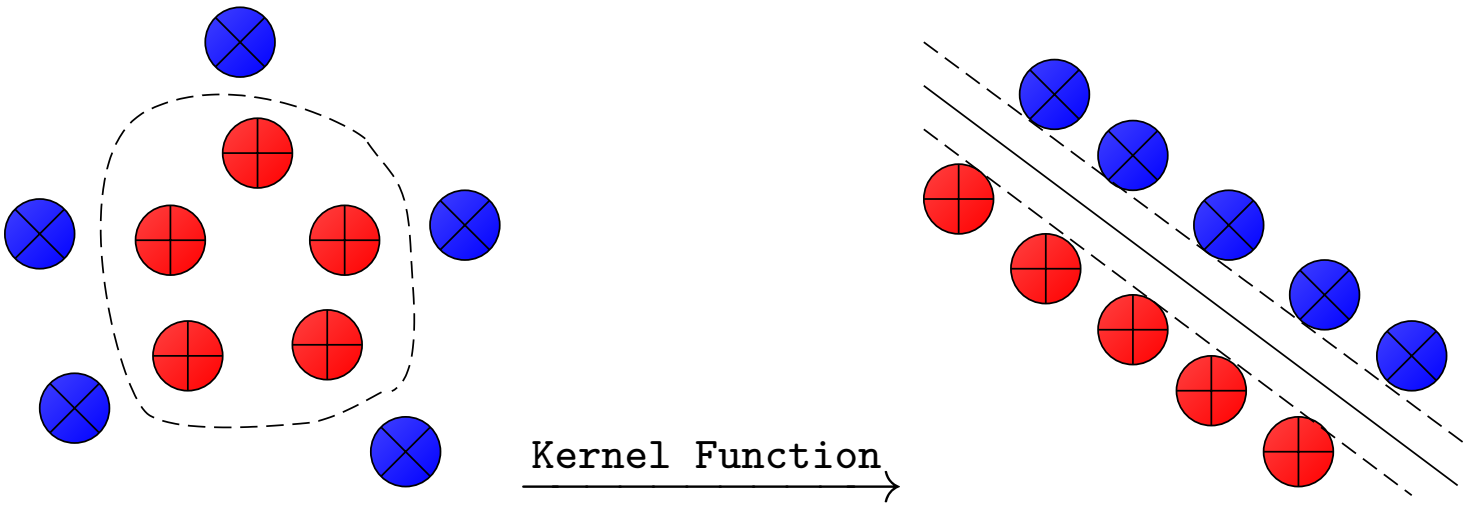
Support Vector Machine

- A supervised machine learning classification technique
- Models a feature space constructed using a set of vectors
- Each vectors has a set of attributes and a category
- Attempts to linearly separate vectors



Support Vector Machine – cont.

- Can work for *many*-group classification
- Can work on non-linearly separable data using kernel functions

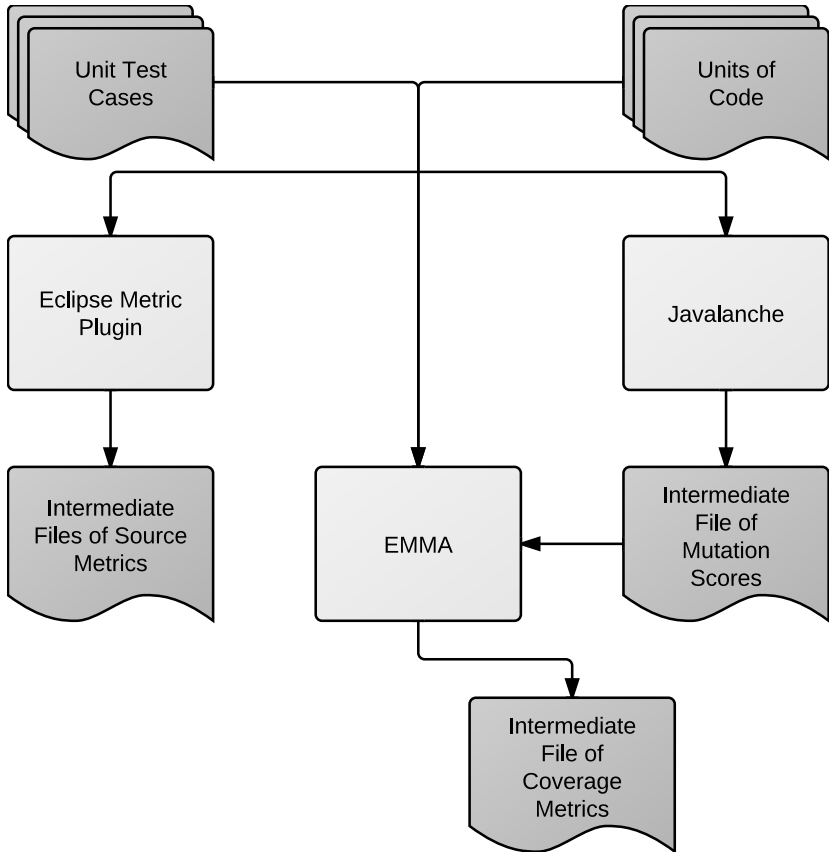


Metrics

- Measurements that describe structural and behavioral properties of a software system – We believe these properties affect the mutation score of source code units
- The combination of source code and test metrics is fairly unique to our research.

Data Collection

- Input: Units of test cases and units of source code
  1. Collect mutation scores using `Javalanche`[a]
  2. Collect source code metrics using `Eclipse Metrics Plugin`[b]
  3. Collect coverage metrics using `EMMA`[c]



. . .

[a] `github.com/david-schuler/javalanche`
[b] `metrics2.sourceforge.net/`
[c] `emma.sourceforge.net/`

Source Code Metrics – Methods

Description	Scope
Method lines of code	Method
Nested block depth	Method
McCabe cyclomatic complexity	Method
Number of parameters	Method

Source Code Metrics – Classes

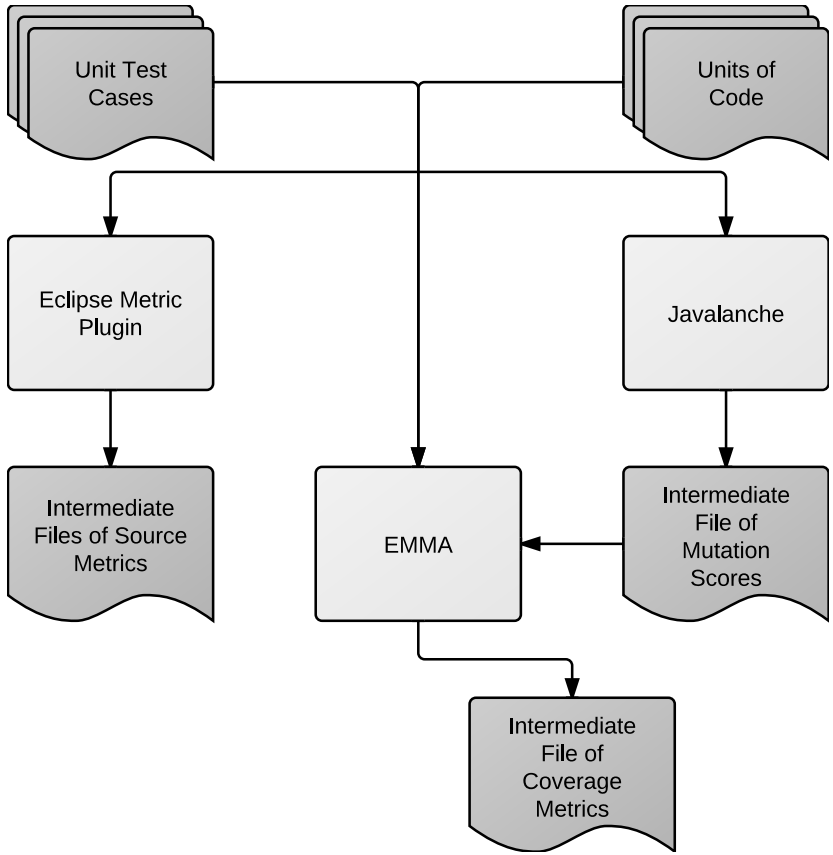
Description	Scope
Number of overridden methods	Class
Number of attributes	Class
Number of children	Class
Depth of inheritance tree	Class
Lack of cohesion of methods	Class
Number of static methods	Class
Number of methods	Class
Specialization index	Class
Weighted method per class	Class
Number of static attributes	Class

Test Code Metrics – Coverage

Description	Scope
Basic blocks covered in code unit	Class/Method
Total basic blocks for code unit	Class/Method

Data Collection [recap.]

- Input: Units of test cases and units of source code
  1. Collect mutation scores using `Javalanche`[a]
  2. Collect source code metrics using `Eclipse Metrics Plugin`[b]
  3. Collect coverage metrics using `EMMA`[c]

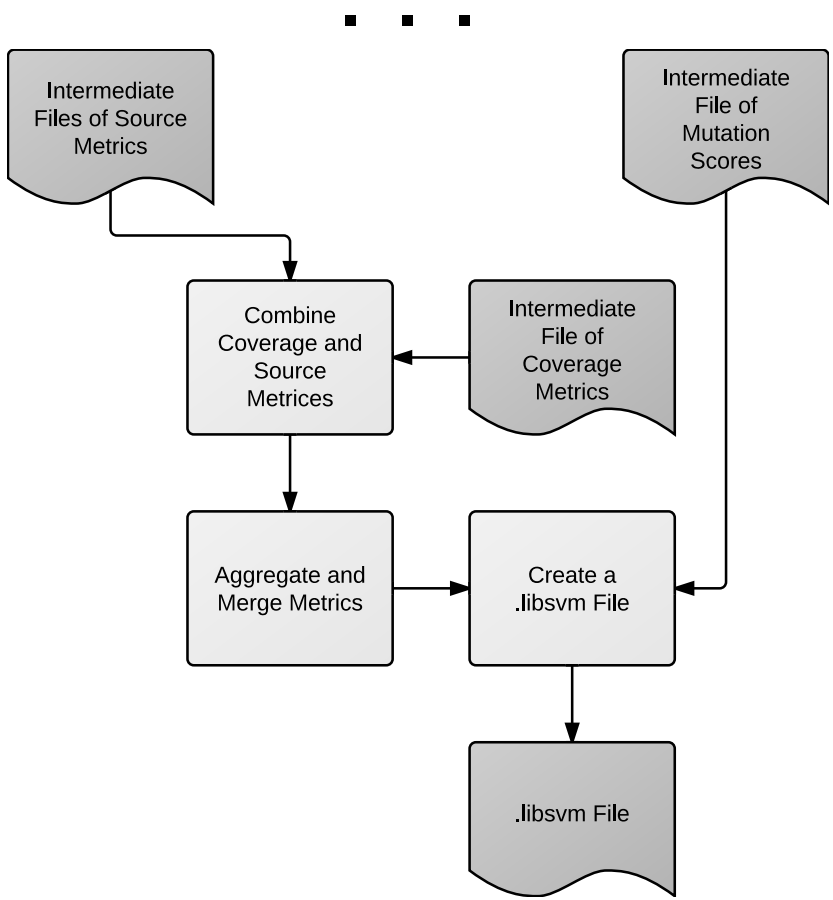


. . .

[a] `github.com/david-schuler/javalanche`
[b] `metrics2.sourceforge.net/`
[c] `emma.sourceforge.net/`

Data Synthesis

- Goal: Category/feature data for source code units
  1. Combine source code and coverage metrics together
  2. Merge source code metrics of the touched tests for each source code unit
  3. Aggregate method-level source code unit metrics into class-level source code units
  4. Create `.libsvm` file using feature/category data



Test Code Metrics – Accumulated Tests

- Each source code unit has a set of associated test cases that touch the unit during test execution.
- We can acquire the summation and average of the associated test cases for each source code unit.

Description	Scope
Method lines of code	Method
Nested block depth	Method
McCabe cyclomatic complexity	Method
Number of parameters	Method

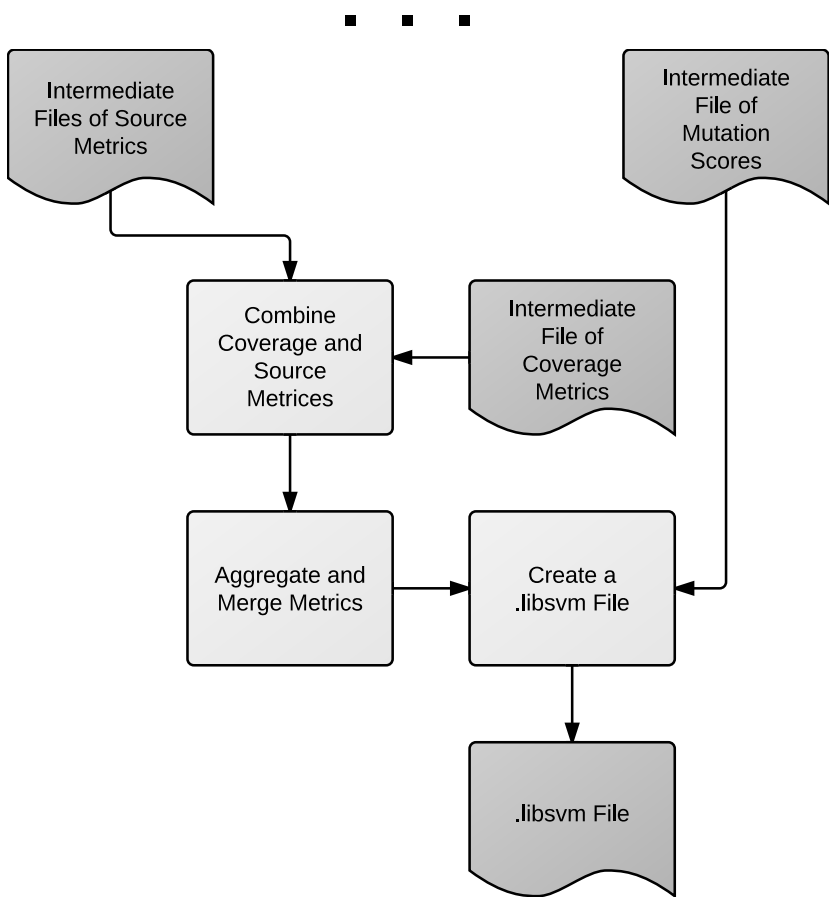
Source Code Metrics – Accumulated Methods

- Class source code units can also have summation and average value of method scope source code metrics

Description	Scope
Method lines of code	Method
Nested block depth	Method
McCabe cyclomatic complexity	Method
Number of parameters	Method

Data Synthesis [recap.]

- Goal: Category/feature data for source code units
  1. Combine source code and coverage metrics together
  2. Merge source code metrics of the touched tests for each source code unit
  3. Aggregate method-level source code unit metrics into class-level source code units
  4. Create `.libsvm` file using feature/category data



.libsvm File

```
1 1:2  2:1 3:1 4:1 5:0 6:0 7:0.0
1 1:2  2:1 3:1 4:1 5:0 6:1 7:3.0
2 1:24 2:3 3:1 4:1 5:1 6:1 7:16.0
2 1:31 2:6 3:3 4:3 5:1 6:1 7:17.0
3 1:1  2:1 3:1 4:1 5:1 6:2 7:16.0
3 1:23 2:7 3:2 4:2 5:0 6:0 7:0.0
...
```

- LIBSVM[a] is a SVM library that requires a specific file format for training and prediction
  - `category attribute_1:value attribute_2:value ...`
- We have class and method `.libsvm` file for predicting and training

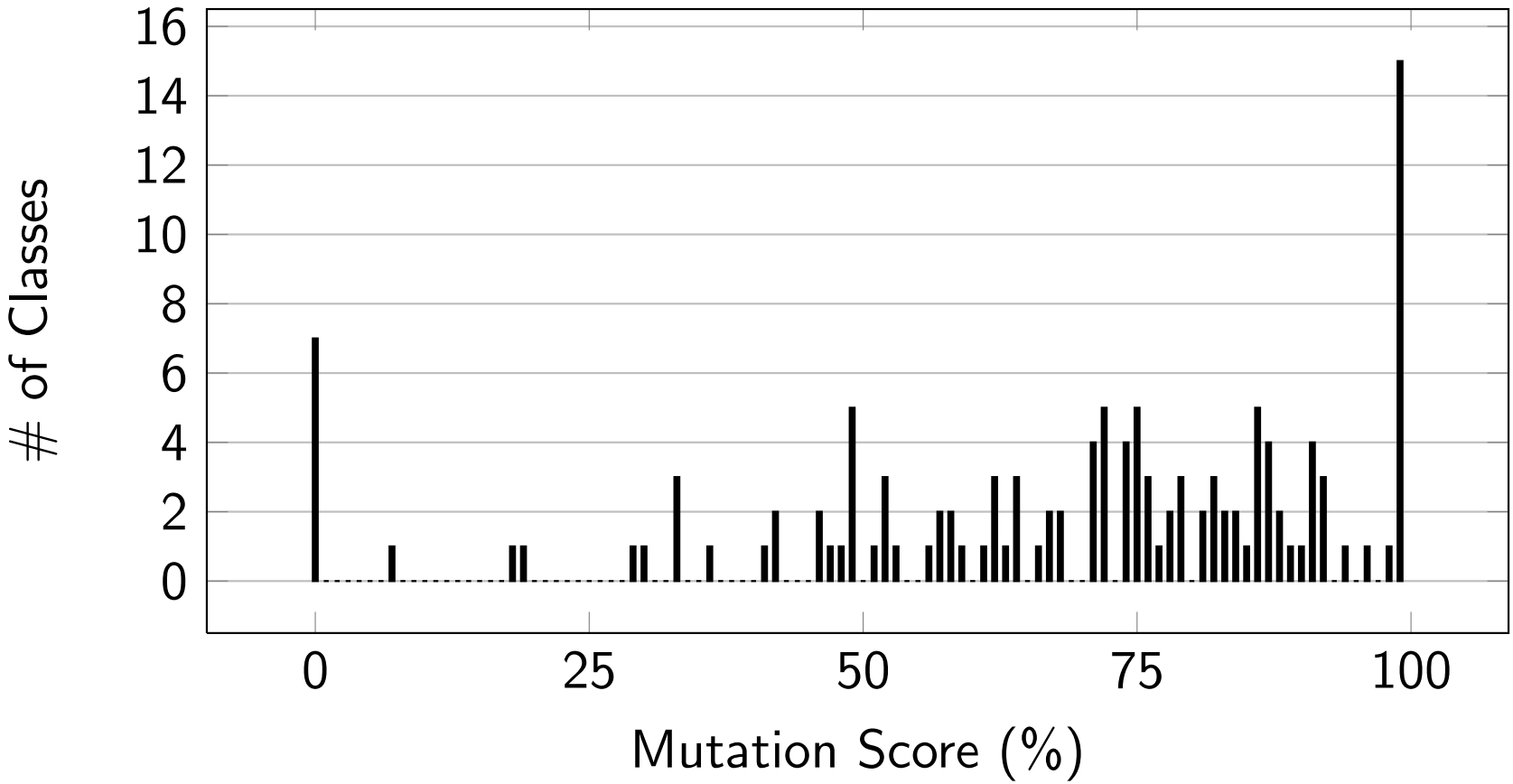
[a] csie.ntu.edu.tw/~cjlin/libsvm/

Case Study – JGAP

- We present initial results on an open source project – JGAP (Java Genetic Algorithm Package).
- JGAP source code has 28975 SLOC, 3017 methods, 415 classes.
- JGAP test code has 19556 SLOC, 1626 methods, 180 classes.
- In total JGAP has 1412[a] JUnit test cases.

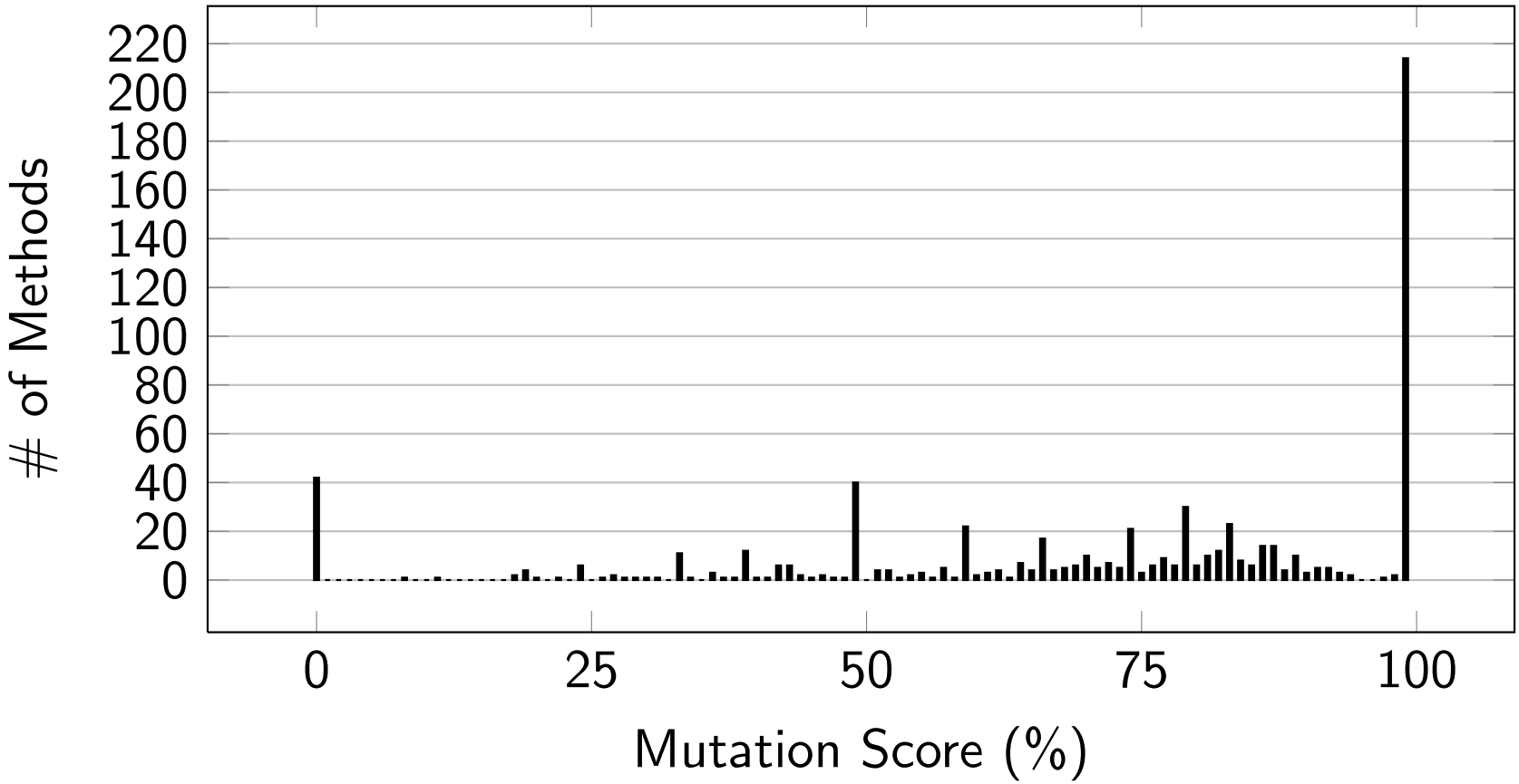
[a] JGAP has 1412 JUnit test cases in total, however 25 of the tests caused errors in the `Javalanche` tool and were removed.

JGAP – Class Mutation Score Distribution



- Collected 127 class-level data points.

JGAP – Method Mutation Score Distribution



- Collected 695 method-level data points.

JGAP – Prediction Categories

Category	Class Mutation Score	Method Mutation Score
low	0.00% – 62.75%	0.00% – 66.66%
medium	62.75% – 83.25%	66.66% – 90.90%
high	83.25% – 100.00%	90.90% – 100.00%

- These mutation score ranges were selected to evenly distribute the training data points across categories.

JGAP – Cross Validation Accuracy

Set	Class Accuracy	Method Accuracy
Source Metrics	53.54%	48.77%
Coverage Metrics	49.61%	47.63%
Sum/Avg Test Metrics	45.67%	49.78%
Sum/Avg Source Metrics	54.33%	33.96%
All Metrics	**58.27%**	**54.82%**

- This validates our initial intuition that we need both source code and test suite metrics to predict mutation score

More Attributes for Feature Data

- We are now considering new attributes for feature data:
  - Number of Tests Cases for each source code unit.
  - Number of Mutant Types for each source code unit.
  - Number of Total Mutants for each source code unit.
- We also want to optimize the current set of attributes

More Data, More Questions

Program	SLOC	Test SLOC	Test Cases
jgap (3.6.1)	28975	19694	1355
joda-time (2.0)	27139	51428	3866
commons-lang (3.1)	19499	33332	2050
logback-core (1.0.3)	12118	8145	286
openfast (1.1.0)	11646	5587	322
joda-primitives (1.0)	11157	6989	1810
jsoup (1.6.2)	10949	2883	319
barbecue (1.5-beta1)	4837	2910	225

- In general can we predict within projects?

More Data, More Questions

Program	SLOC	Test SLOC	Test Cases
jgap (3.6.1)	28975	19694	1355
joda-time (2.0)	27139	51428	3866
commons-lang (3.1)	19499	33332	2050
logback-core (1.0.3)	12118	8145	286
openfast (1.1.0)	11646	5587	322
joda-primitives (1.0)	11157	6989	1810
jsoup (1.6.2)	10949	2883	319
barbecue (1.5-beta1)	4837	2910	225

- Can we predict across projects?

Predicting Mutation Score Using Source Code and Test Suite Metrics

RAISE 2012

Kevin Jalbert and Jeremy S. Bradbury

{kevin.jalbert,jeremy.bradbury}@uoit.ca

Software Quality Research Group (sqrg.ca)
University of Ontario Institute of Technology
Oshawa, Ontario, Canada

June 5^{th}, 2012