EECS4313
Software Engineering Testing
Assignment 2 (150 points), Version 1

Instructors: Song Wang
Release Date: Feb 8, 2020

Due: 11:59 PM, Tuesday, March 5, 2020

Check the Amendments section of this document regularly for changes, fixes, and clarifications.

Ask questions on the course forum on the Moodle site.

Policies

Your (submitted or un-submitted) solution to this assignment (which is not revealed to the public) remains the property of the EECS department. Do not distribute or share your code in any public media (e.g., a non-private Github repository) in any way, shape, or form. The department reserves the right to take necessary actions upon found violations of this policy.

- You are required to work on your own for this assignment. No group partners are allowed.
- When you submit your solution, you claim that it is solely your work. Therefore, it is considered as an violation of academic integrity if you copy or share any parts of your Java code during any stages of your development.
- We will conduct plagiarism detection on all students' code.
- When assessing your submission, the instructor and TA may examine your code, and suspicious submissions will be reported to the department/faculty if necessary. We do not tolerate academic dishonesty, so please obey this policy strictly.
- Emailing your solutions to the instruction or TAs will not be acceptable.

Submitting Your Work

Create subfolders with names "pi", "pii" for your solutions to part I and Part II.

Inside "pi", create subfolders with names "a", "b", and "c" for your solutions to questions a, b, and c.

Inside "pii", create subfolders with names "a" and "b" for your solutions to questions a and b.

```
zip -r EECS4313_A2.zip EECS4313_A2
submit 4313 a2 EECS4313_A2.zip
```

Amendments

- so far so good!

Part I: Build Rule-Based Bug Detectors

(a) Inferring Likely Invariants for Bug Detection (50 points)

Take a look at the following contrived code segment.

```
void scope1() {
A(); B(); C(); D();
}
void scope2() {
A(); C(); D();
}
void scope3() {
A(); B(); B();
}
void scope4() {
B(); D(); scope1();
}
void scope5() {
B(); D(); A();
}
void scope6() {
B(); D();
}
```

We can learn that function `A` and function `B` are called together three times in function `scope1`, `scope3`, and `scope5`. Function `A` is called four times in function `scope1`, `scope2`, `scope3`, and `scope5`. We infer that the one time that function `A` is called without `B` in `scope2` is a bug, as function `A` and `B` are called together 3 times. We only count `B` once in `scope3` although `scope3` calls `B` two times.

We define *support* as the number of times a pair of functions appears together. Therefore, $support(\{A, B\})$ is 3. We define $confidence(\{A, B\}, \{A\})$ as $\frac{support(\{A,B\})}{support(\{A\})}$, which is $\frac{3}{4}$. We set the threshold for support and confidence to be $T\_SUPPORT$ and $T\_CONFIDENCE$, whose default values are 3 and 65% (can be configurable). You only print bugs with confidence $T\_CONFIDENCE$ or more and with pair support $T\_SUPPORT$ times or more. For example, function `B` is called five times in function `scope1`, `scope3`, `scope4`, `scope5`, and `scope6`. The two times that function `B` is called alone are not printed as a bug as the confidence is only 60% $\left(\frac{support(A,B)}{support(B)} = \frac{3}{5}\right)$, lower than the $T\_THRESHOLD$, which is 65%. Please note that $support(A, B)$ and $support(B, A)$ are equal, and both 3.

Perform intra-procedural analysis. For question (a), we do not expand `scope1` in `scope4` to the four functions `A`, `B`, `C`, and `D`. Match function names only. Do not consider function parameters. For example, `scope1()` and `scope1(int)` are considered the same function.

The sample output with the default support and confidence thresholds should be:

```
bug: A in scope2, pair: (A, B), support: 3, confidence: 75.00%
bug: A in scope3, pair: (A, D), support: 3, confidence: 75.00%
bug: B in scope3, pair: (B, D), support: 4, confidence: 80.00%
bug: D in scope2, pair: (B, D), support: 4, confidence: 80.00%
```

Generate call graphs using BCEL framework. The Byte Code Engineering Library (Apache Commons BCEL) is intended to give users a convenient way to analyze, create, and manipulate (binary) Java class files (those ending with `.class`). Classes are represented by objects which contain all the symbolic information of the given class: methods, fields and byte code instructions, in particular. Such objects can be read from an existing file, be transformed by a program (e.g. a class loader at run-time) and written to a file again. An even more interesting application is the creation of classes from scratch at run-time. The Byte Code Engineering Library (BCEL) may be also useful if you want to learn about the Java Virtual Machine (JVM) and the format of Java .class files.

The details of `BCEL` are here:

`https://commons.apache.org/proper/commons-bcel/manual/bcel-api.html`

Given a bytecode file, we can use BCEL to generate call graphs and analyze the textual call graphs to generate function pairs and detect bugs. **The *java-callgraph* (`github.com/gousiosg/java-callgraph`) tool** (an application of `BCEL`) is given to you for generating call graphs from bytecode. You can find the tool in `/pi`.

Instruction of run "java-callgraph" tool:

You can use the following command to generate call graph for a given `jar` file in text format.

```
java -jar javacg-0.1-SNAPSHOT-static.jar {path-to-a-jar-file}
```

e.g., to generate call graph for project "commons-math3-3.6.1.jar":

```
java -jar javacg-0.1-SNAPSHOT-static.jar proj/commons-math3-3.6.1.jar
```

Given the following Java source file:

```
public class Test_Null {
        public static void main(String [] args){
                String str = null;
                System.out.println(str.toLowerCase());
        }
}
```

A call graph generated from the bytecode of the above Java code in text format is shown as follows:

```
C:Test_Null Test_Null
C:Test_Null java.lang.Object
C:Test_Null java.lang.System
C:Test_Null java.lang.String
C:Test_Null java.io.PrintStream
M:Test_Null:<init>() (O)java.lang.Object:<init>()
M:Test_Null:main(java.lang.String[]) (M)java.lang.String:toLowerCase()
M:Test_Null:main(java.lang.String[]) (M)java.io.PrintStream:println(java.lang.String)
```

The following explanation should help you understand the call graph:

- **Format for classes (starting with "C"):**

  `C:class1 class2`

  This means that some method(s) in `class1` called some method(s) in `class2`.

- **Format for methods (starting with "M"):**

  `M:class1:<method1>(arg_types) (typeofcall)class2:<method2>(arg_types)`

  The line means that `method1` of `class1` called `method2` of `class2`.

- A valid method call sequence is: **[java.lang.String:toLowerCase, java.io.PrintStream:println]**, please ignore the `java.lang.Object:<init>` and `main()` methods as these methods are essential initialization methods or entry points of Java applications.

Performance. We will evaluate the performance and scalability of your program on a pass/fail basis. The largest program that we test will contain up to 50k nodes and 100k edges. Each test case will be given a timeout of 10 minutes. A timed-out test case will receive zero points. We do not recommend using multi-threading because it only provides a linear gain in performance.

Hints:

- (Very Important!) Do not use string comparisons because it will greatly impact the performance. Use a hashing method to store and retrieve your mappings.
- Focus rules of method call pairs, i.e., the length of a rule is two.
- Minimize the number of nested loops. Pruning an iteration in the outer loop is more beneficial than in the inner loop.

Your tasks:

- Implement your code for inferring rules of **function pairs** and detecting bugs with these rules in the given python file: `detector.py` (in folder `/pi`).
- Your `detector.py` takes three parameters: `-jar <path_to_jar> -sup <support> -c <confidence>` , an example is as follows:

  ```
  detector.py -jar proj/opennlp-tools-1.9.2.jar -sup 5 -c 0.75
  ```

- The output will be the bugs (i.e., violations) detected by rules inferred with the following format:

  ```
  bug: X in M, pair: (X, Y), support: S, confidence: C
  ```

  Examples are:

  ```
  bug: B in method2, pair: (A, B), support: 3, confidence: 75.00%
  bug: D in method3, pair: (B, D), support: 4, confidence: 80.00%
  ```

(b) Finding and Explaining False Positives (30 points)

Table 1: Details of the experimental subjects.

Project	version	description	#method	#class
commons-math3	3.6.1	a library of lightweight, self-contained mathematics and statistics components	36K	11K
solr	8.4.1	an open-source enterprise-search platform , written in Java, from the Apache	49K	21K
weka	3.6.1	open source machine learning software	164K	29K
deeplearning4j	1.0.0	a deep learning programming library written for Java	0.8K	0.3K
opennlp	1.9.2	a machine learning based toolkit for the processing of natural language text	29K	11K
mahout	0.9	a distributed linear algebra framework	25K	13K

You will apply your code to the latest versions of real-world projects listed in table 1 (you can find their jar files in folder `/proj`). Examine the output of your code for (a) (i.e., `detector.py`) on the each of the six projects. The values of *T_SUPPORT* and *T_CONFIDENCE* are 3 and 65%.

Do you think all the reported bugs are real bugs? There are some false positives. A false positive is where a line says "bug ..." in your output, but there is nothing wrong with the corresponding code. Write down two different fundamental reasons for as to why false positives occur in general.

Identify 2 pairs of functions that are false positives for each project. Explain why you think each of the two pairs that you selected are false positives. The two pairs should have a combined total of at least 5 "bug ..." line locations (e.g., first pair can have 4 and second pair can have 6). For example, the following sample output contains 4 locations

regarding 3 pairs: (A, B) (A, D) and (B, D). You do not have to provide explanations for all 5 locations, but you will want to use at least one location to discuss each pair. Briefly explain why the pair is a false positive with reference to the source code.

```
bug: A in scope2, pair: (A, B), support: 3, confidence: 75.00%
bug: A in scope3, pair: (A, D), support: 3, confidence: 75.00%
bug: B in scope3, pair: (B, D), support: 4, confidence: 80.00%
bug: D in scope2, pair: (B, D), support: 4, confidence: 80.00%
```

The source code of each experiment projects has been provided under the folder `proj/source/`. You can find more detailed context information for each reported bugs. This will also help you explain the reason of false positives.

If you find new bugs (bugs that have not been found by other people yet), you may receive bonus points. Check the corresponding bug databases of these projects, e.g., commons-math3[1] and solr[2], to make sure that the bug you found has not been reported already. Clearly explain why you think it is a bug and provide evidence that it is a new bug.

(c) Inter-Procedural Analysis. (20 points)

One solution to reduce false positives is inter-procedural analysis. For example, one might expand the function `scope1` in the function `scope4` to the four functions `A`, `B`, `C`, and `D`. Implement this solution by adding an optional fourth command line parameter, i.e., `-inp <true:use inter-procedural analysis; false:no inter-pro analysis>`, to `detector.py`. We leave the algorithm's implementation details up to you.

Write a report (up to 2 pages) to describe your algorithm and what you found. Run an experiment on your algorithm and report the details. Provide a concrete example to illustrate that your solution can do better than the default algorithm used in (a). We will read your report and code. If you only submit a report, you will receive at most 50% of the points for this part (c). Make sure your code is well documented and your report is understandable.

[1] `https://issues.apache.org/jira/projects/MATH/issues`
[2] `https://issues.apache.org/jira/projects/SOLR/issues/`

Part (II) Using Static Bug Detection Tools: FindBugs and Pylint

For this part of the project you will be using the static code analysis tool FindBugs (`http://findbugs.sourceforge.net/`) to find defects in an open source project and then using Pylint (`https://www.pylint.org/`) to find possible defects in your own code, i.e., `detector.py` from **Part I**.

(a) Triage Bugs from Apache Common-math3 (30 points)

In this task, you are to triage warning messages resulting from running FindBugs static analysis on Apache commons-math3 (version 3.6.1). Your task is to triage these warnings. We select 30 warnings for you to triage (in **pii/commons-math3-warning.txt**). Each warning contains a pattern and a violation to the pattern, an example is as follows:

```
Eq: org.apache.commons.math3.linear.RealVector.equals(Object) is unusual At RealVector.java:[line 1096]
```

Eq is the abbreviation of a bug pattern about method `equals()`, for details of each bug pattern, read the documentation (see `http://findbugs.sourceforge.net/bugDescriptions.html`). Following the bug pattern **Eq** is the detected bug location, i.e., **org.apache.commons.math3.linear.RealVector.equals(Object)**. You can use the line number **line 1096** to find more context information in the source file of class **RealVector.java**. The source code of each experiment projects has been provided under the folder `proj/source/`.

Triage each warning by classifying it as one of the following: *False Positive*, *Intentional*, or *Bug*. Write your classification and explanations directly in your report. Below are descriptions of each triage classification:

- **False Positive**: The warning does not indicate a fault or deficiency.
- **Intentional**: You believe the warning points to code that the developer **intentionally** created that is incorrect (contains a fault) or considered bad practice.
- **Bug**: The warning points to a defect (a fault or bad practice) in the code.

If you classify the warning as *False Positive*, provide an explanation as to why you believe this is the case. If you classify the warning as *Intentional*, explain how the code can be re-factored to remove the warning, or explain why it should be left as-is. If you classify the warning as bug, provide the faulty lines (1-2 lines is often enough, including the file name and line numbers) and a description of a proposed bug fix.

The difference between False Positive/Intentional and Intentional/Bug is not always clear, so there is not necessary one unique good classification. Make sure you explain clearly your choice!

The report for this part should be no more than 4 pages. Be concise.

(b) Analyzing Your Own Code (20 points)

Run Pylint on your own code from part (I) (a). See instructions on `https://www.pylint.org/` to analyze your code. You learn more from having the tool find defects in code that you wrote, fixing the problem and seeing the defect go away. Discuss two of the bugs detected by Pylint using up to 1 page. If Pylint finds no bugs, what are the possible reasons?