EECS4313 Software Engineering Testing
Description of Defect Prediction Data for Project, Version 1

Instructor: Song Wang
Release Date: March 12, 2020

1 Defect Data Description

Table 1: Evaluated projects for change-level defect prediction. **Lang** is the programming language used for the project. **LOC** is the number of the line of code. **First Date** is the date of the first commit of a project, while **Last Date** is the date of the latest commit. **Changes** is the number of changes collected in this work. **TrSize** is the average size of training data on all runs. **TSize** is the average size of test data on all runs. **ABR** is the average buggy rate. **NR** is the number of runs for each subject.

Project	Lang	LOC	First Date	Last Date	Changes	TrSize	TSize	ABR (%)	# NR
Xorg	C	1.1M	1999-11-19	2012-06-28	46K	1,756	6,710	14.7	6
JDT	Java	1.5M	2001-06-05	2012-07-24	73K	1,367	6,974	20.5	6
Lucene	Java	828K	2010-03-17	2013-01-16	76K	1,194	9,333	23.6	6
Jackrabbit	Java	589K	2004-09-13	2013-01-14	61K	1,118	8,887	37.4	6

This dataset is collected from four open-source projects, i.e., Xorg, Jdt (from Eclipse), Lucene, and Jackrabbit. They are large and typical open source projects covering operating system, database management system. These projects have enough change history to build and evaluate change-level defect prediction models. Table 1 shows the evaluated projects for change-level defect prediction. Each project have more multiple runs, each run contains a training dataset and a test dataset, which are collected during different time periods. For example, Lucene has 6 runs, **you are expected to train and test you model on each of the 6 runs, and use the average performance to evaluate your prediction models.** The LOC and the number of changes in Table 1 include only source code (C and Java) files[1] and their changes.

Each folder of each project contains both the training and test datasets for each run. Each change has an unique change id and the label (buggy or clean). We have already collected the source code for each change, which is stored in **/data/*proj*/patch.zip**.

We also provide the mapping between each change and its commit. **We consider the modification on a file as a change**. In a commit, there may exist multiple changes. For example, to fix a bug, a developer changed three files: `f1`, `f2`, and `f3` and then s/he submitted them together in a commit. The mapping information is in **/data/*proj*/*proj*_change_id_mapping.csv**.

1.1 How to Collect Features

Bag-of-Words: Take the first change in the training data of **folder 0** from Lucene, i.e., **9007** as an example. The corresponding source code of this change is **lucene-9007.patch** in **/data/lucene/patch.zip**.

```
diff --git a/solr/src/java/org/apache/solr/handler/component/QueryComponent.java b/solr/src/java/org/
    apache/solr/handler/component/QueryComponent.java
index 0415f29..96f3893 100644
--- a/solr/src/java/org/apache/solr/handler/component/QueryComponent.java
+++ b/solr/src/java/org/apache/solr/handler/component/QueryComponent.java
```

[1] We include files with these extensions: .java, .c, .cpp, .cc, .cp, .cxx, .c++, .h, .hpp, .hh, .hp, .hxx and .h++.

```
@@ -217,14 +217,8 @@ public class QueryComponent extends SearchComponent
           for (String groupByStr : funcs) {
             QParser parser = QParser.getParser(groupByStr, "func", rb.req);
             Query q = parser.getQuery();
-            SolrIndexSearcher.GroupCommandFunc gc;
-            if (groupSort != null) {
-              SolrIndexSearcher.GroupSortCommand gcSort = new SolrIndexSearcher.GroupSortCommand();
-              gcSort.sort = groupSort;
-              gc = gcSort;
-            } else {
-              gc =  new SolrIndexSearcher.GroupCommandFunc();
-            }
+            SolrIndexSearcher.GroupCommandFunc gc = new SolrIndexSearcher.GroupCommandFunc();
+            gc.groupSort = groupSort;

             if (q instanceof FunctionQuery) {
               gc.groupBy = ((FunctionQuery)q).getValueSource();
```

You can collect the **Bag-of-Words** features for each change by using the code of each change we provided. Specifically, you need first to tokenize each change by filtering unnecessary symbols, then collect all the frequent tokens (e.g., the frequency is larger than 3) from all the changes from a folder (both the training and test datasets). Note that, you may want to remove the keywords from a specific program language, e.g., `int, double, boolean, class, public, void, private`, etc., as these tokens often are not related to program errors could be noises. Meanwhile, we keep the control flow related keywords, e.g, `if, else, for, while`, etc. Finally, building the bag-of-words model for changes.

Useful information about the **Bag-of-Words** model can be found here:

`https://en.wikipedia.org/wiki/Bag-of-words_model`

2 Frequent Questions

- **Q**: How to evaluate a specific classifier across all runs in a project?

  We focus on evaluating the performance of **predicting buggy instances.** For evaluating the performance of the classifier on a project, you could accumulate the FalsePositive, False Negatives, TruePositives, TrueNegatives across folders and then calculate the Precision, Recall, and F1. **We use F1 to measure the performance of a classifer.**

- **Q**: To what extend should we tune a classifier?

  You are expected to explore the performance of different values for each parameter (if the classifier has parameters). If a parameter could be arbitrary numerical values, try at least ten discrete values (e.g., 1,2,5,8,10,20,30,50,80, and 100).

- **Q**: Are we allowed to use other libs other than `scikit-learn`?

  Yes, please also submit the machine learning libs you used.