1 Falessi

1.1 "Leveraging Defects Lifecycle for Labeling Defective Classes"

Software Defect Prediction (SDP) is one of the most useful and cost efficient activity *used to identify software modules that are defect prone and require extensive testing in order to use testing resources efficiently.*

Software defect predictors are based on statistical approach and, particularly, machine learning. However Some research studies adopted recent machine learning techniques such as active/semi-supervised learning to improve prediction performance. Prediction models learned by machine learning algorithms can predict either bug-proneness of source code (classification) or the number of defects in source code (regression).

The software defect prediction process based on **machine learning models** requires some steps:

1. First of all is necessary to generate the **instances** from software archives for our prediction model. Instances can be generated from version control systems, issue tracking systems, e-mail archives, and so on. Each instance represents a system, a software component, a source code file, a class, a function and so on.

   An instance can have several metrics (or features) extracted from the software archives and is labeled with buggy/clean or the number of bugs.

2. After generating instances with metrics and labels, we can apply preprocessing techniques, which are common in machine learning. Preprocessing techniques used in defect prediction studies include **feature selection**, **data normalization**, and **noise reduction**. However preprocessing is an optional step.

3. Finally we have to train a prediction model in such a way it can predict whether a new instance has a bug or not. *The prediction for bug-proneness (buggy or clean) of an instance stands for binary classification, while that for the number of bugs in an instance stands for regression.*

Clearly source code metrics measure how source code is complex and the main rationale of the source code metrics is that source code with higher complexity can be more bug-prone.

Process metrics are extracted from software archives such as version control systems and issue tracking systems.

To measure defect prediction results by classification models, we should consider the following prediction outcomes first:

True Positive (TP) buggy instances predicted as buggy.

False positives (FP) clean instances predicted as buggy.

True negative (TN) clean instances predicted as clean.

False negative (FN) buggy instances predicted as clean.

With these outcomes, we can define the following measures, which are mostly used in the defect prediction literature.

Precision :

$$\frac{TP}{TP + FP} \tag{1}$$

Recall Recall, also know as **probability of detection** (**PD**) or **true positive rate** (**TPR**), measures correctly predicted buggy instances among all buggy instances:

$$\frac{TP}{TP + FN} \tag{2}$$

F-measure F-measure is a harmonic mean of precision and recall:

$$\frac{2 \times (Precision \times Recall)}{Precision + Recall} \tag{3}$$

Although bug prediction is an important tool, it was not possible to study the origin of bugs in large-scale scenarios until the introduction of the **SZZ algorithm**, which was proposed by Sliwerski, Zimmermann and Zeller - hence the acronym.

The SZZ algorithm *traces back the code history to find changes that are likely to introduce bugs*, i.e., the so-called **bug-introducing changes**.

However, SZZ is not without limitations because it may produce inaccurate data by not recognizing that bugfix changes may contain interleaved refactorings, since code refactoring does not directly fix a bug. Similarly, SZZ may erroneously flag refactoring changes as bug-introducing changes.

SZZ is an algorithm used to identify bug-introducing changes. In order to identify bug-introducing changes, the SZZ algorithm starts analyzing the bug-fix changes, which are changes that are known to fix a bug that is reported on an Issue Tracking System (e.g. JIRA and Bugzilla).

Using change logs provided by VCSs, the SZZ algorithm identifies a bug ID and the bug-fix change.

Next, SZZ performs a `diff` operation between the bug-fix change and a previous change to identify how the bug was fixed.

Finally, to locate the bug-introducing change, SZZ traces back in code history (e.g., using the `git blame` function) to find the change that introduced the bug, the bug-introducing change.