

ISW2 Project A.A. 2019-2020

Andrea Graziani (0273395)
andrea.graziani93@outlook.it
Università degli Studi di Roma "Tor Vergata"
Rome, Italy

KEYWORDS

Testing, Equivalence Class Partitioning, Boundary Value Analysis, Mutation Testing

1 INTRODUCTION

Why are some programs more defect-prone than others?

This is one of the central questions of software engineering. To answer it, we must first know *which* programs or software modules are more failure-prone than others, afterwards, with this knowledge, we can search for properties of the program that are correlated with defect density. How to do that?

Let's start now with some definitions. A **software defect** can be generally considered as an error, flaw, bug, mistake, failure, or fault in a computer program or system that may generate an inaccurate or unexpected outcome, or precludes the software from behaving as intended [4].

Software defects always incur cost in terms of time because identifying and rectifying defects is one of the most time consuming and expensive software processes [4].

Although it's not practically possible to eliminate each and every defect from a software product, we can reduce their adverse effects exploiting **software defects prediction** techniques, which aim is to identify software modules that are defect prone, in order to reduce the cost of testing activities and code review by letting developers focus on specific artefacts [1].

How to build a software defect prediction model?

The building of a software prediction model based on **machine learning** techniques passes through several steps which we will describe them in following sections.

1.1 Dataset building

Firstly, is necessary to generate a data-set to provide an appropriate input for our prediction model, that is we need to build a set of **instances**. In our context, each data-set's instance represents a **source code file** belonging to an open-source software project and characterized by its values on a fixed, predefined set of **features**, or **attributes**.

Clearly, in order to build an useful software predictor, is preferable to collect instances whose features are correlated to the defect-proneness of the software module they belong. Fortunately, there are many studies about software defect-proneness which have identified several features (sometimes called also **metrics**) correlated with defect density of a software module.

For example, Zimmermann et al. [7] showed that the number of past defects has the highest correlation with number of future defects. Nagappan and Ball [3] report that *code churn* is a very powerful metric for predict software defect density, while Gyimothy et al. [2] showed the same with *lines of code* (LOC) metric.

Defect-proneness correlated features are extracted from *history data* of software module, including software archives and tools like *version control systems* and *issue tracking systems*, because they are generally assumed as a good source for future fault prediction.

1.2 Labelling activity

Since our aim is to build a software defect predictor using a *supervised learning scheme*, is necessary to perform the so-called *labelling* task, that is an activity according to which one or more informative tags are assigned to each unlabelled instance of a given dataset.

In fact, when a supervised learning scheme is used, the goal is to learn a function that maps an input to an output based on example input-output pairs. In other word, each data-set instance T is represented as a pair (x_d, y) where x_d is a d -dimensional input object while y is the desired output value. In this way, a supervised learning algorithm, analysing labelled training data, can produce an inferred function, which can be used for mapping new instances.

In our context, to build a properly data-set, we need to label each instance as *buggy* or *not buggy*.

In order to do that during the development of our predictor, we have exploited all information retrievable both from JIRA¹, one of the most popular issue tracking systems, and git², the most common version control systems.

Listing 1: Dataset build logic

```
public void buildProjectDataset() {  
    collectReleases();  
    collectFilesBelongingToEachRelease();  
  
    IssueRegistry issueRegistry = this.issueTrackingSystem.getIssuesRegistry(this.project.name);  
  
    calculateDefectiveFileProportion(issueRegistry.issuesWithAffectedVersions);  
    searchForDefectiveFile(issueRegistry.issues);  
  
    collectFileMetadataOfEachRelease();  
}
```

First of all, it is necessary to identify all project's releases.

To do that, we have simply queried Jira to obtain all releases data, including names, release dates and so on. Subsequently, we have exploited git tags inside commits messages to identify the commit associated to each release. However, when sometimes git tags were not available, we have identified aforementioned commits exploiting git log command with --before [date] options. collectReleases method, inside ProjectDatasetBuilder³ class, contains the implementation of this step.

To discover all known bugs, we have queried once again Jira to obtain the identifiers of all issues with type "Bug" and status "Closed" (or "resolved"). We remember that is possible to find

¹<https://www.atlassian.com/it/software/jira>

²<https://git-scm.com/>

³datasetbuilder.ProjectDatasetBuilder

the implementation of this query procedure inside `getIssuesRegistry` method into Jira class ⁴.

Is very important to precise that scientific literature reports that most of the projects have more than 25% of defects whose issue report contained no "Affected Version" (AV). Moreover, the vast majority of all bugs (51%) resulted in not having or having an unreliable AV. Therefore, if information about the AV for a bug is available in Jira, we will consider the earliest AV in Jira to be the release that introduced the bug, otherwise we have adopted proportion method.

However, the creation of such datasets made in this way is far from being perfect because scientific literature reports that most of the projects have more than 25% of defects whose issue report contained no AV. Moreover, the vast majority of all bugs (51%) resulted in not having or having an unreliable AV.

1.3 Preprocessing

After generating our data-set, according to project specifications, we have also applied a preprocessing technique, commonly used in machine learning, called **feature selection** or **attribute selection**.

1.3.1 Introduction.

As known, most machine learning algorithms are designed to learn which are the most appropriate attributes to use for making their decisions. For example, decision tree methods choose the most promising attribute to split on at each point and should never select irrelevant or unhelpful attributes [6].

Although, theoretically, having more attributes give us a more discriminating power, in practice, adding irrelevant attributes to a dataset, often performance of machine learning systems degrades [6].

Deleting unsuitable attributes using any feature selection algorithm, is possible to **improve learning algorithms performances**, although, using specific data-sets, is possible that these techniques **can require extensive computations**, degrading overall performances. This is the reason according to which, as in many others machine learning situations, trial and error, using specific source of data, is the best choice [6].

How to select relevant features?

Clearly, the best way to select relevant attributes is *manually*, based on a deep understanding of the learning problem. however, fortunately, automatic methods exist too. In fact, for our analysis, according to project specifications, we have adopted **Best-first** search method.

This scheme adopts a *greedy approach* to select attribute subset that is most likely to make better predictions. Its peculiarity it that it does not just terminate when the performance starts to drop but keeps a list of all attribute subsets evaluated so far, sorted in order of the performance measure, so that it can revisit an earlier configuration instead [6].

1.3.2 Results analysis.

Comparing results derived from our dataset processed using Best

First search method with those processed without feature selection algorithms, we have noted that Best First method is capable to **improve the accuracy of specific classifiers**.

In fact, as you can see from **fig. 1**, containing prediction evaluation of BookKeeper dataset, when a *Naive Bayes* classifier is used (without any sampling scheme), exploiting Best First method, we have obtained following results:

- *Recall* parameter increased by 27.72%, passing from a value equal to 0.440 to 0.562.
- *Kappa* parameter increased by 40.22%, from 0.179 to 0.251.
- *ROC* parameter increased by 2.47%, from 0.686 to 0.703.
- *Precision* parameter decrease by 1.94%, from 0.630 to 0.618.

In summary, our experiments show that Naive Bayes classifier improves its accuracy when Best First is used.

However, using others classifiers, we have obtained very different results. For example, when **Random Forest** classifier is used, our experiment results show that:

- *Recall* parameter decreased by 12,5%, passing from a value equal to 0.612 to 0.544.
- *Kappa* parameter decreased by 25.28%, from 0.446 to 0.356.
- Both *ROC* and *Precision* parameters decreased by 2.87% and 4.84% respectively.

IBK classifier shows results very similar to those of Random Forest too.

1.3.3 Sampling.

As known, performance of a software defect prediction model depends heavily on the data on which it was trained. In fact, defect prediction models that are trained on imbalanced datasets, that is datasets where the proportion of defective and clean modules is not equally represented, are highly susceptible to producing inaccurate prediction models [5]

To mitigate the risk of imbalanced datasets during our analysis, we have adopt following class rebalancing techniques.

Over-sampling It randomly samples with replacement the minority class to be the same size as the majority class. It is also known as up-sampling.

Under-sampling It samples the majority class in order to reduce the number of majority modules to be the same number as the minority class. It is also known as down-sampling.

SMOTE This technique combats the disadvantages of both over-sampling and under-sampling techniques, creating artificial data based on the feature space similarities from the minority class.

1.3.4 Results analysis.

Comparing results derived from our dataset processed using Best First search method with those processed without feature selection algorithms, we have noted that Best First method is capable to **improve the accuracy of specific classifiers**.

As you can see from **fig. 2**, concerning Concerning

- Independently from used classifier, concerning *Recall* parameter, under-sampling technique perform better than all other sampling techniques.

⁴datasetbuilder.datasources.its.jira.Jira

Name	Description
LOC	Lines of code of X .
LOC_TOUCHED	$\sum_R(\text{addedLOC} + \text{deletedLOC})$, where: <ul style="list-style-type: none"> R is the set of all revisions of X. addedLOC represents the number of added code lines while deletedLOC represents the removed ones.
LOC_ADDED	$\sum_R(\text{addedLOC})$
MAX_LOC_ADDED	$\max_R(\text{addedLOC})$.
AVERAGE_LOC_ADDED	$\frac{\sum_R(\text{addedLOC})}{ R }$
NUMBER_OF_REVISIONS	$ R $
NUMBER_OF_FIX	Number of bug fixes which have involved X .
NUMBER_OF_AUTHORS	Number of authors which developed X .
CHURN	$\sum_R(\text{addedLOC} - \text{deletedLOC})$
MAX_CHURN	$\max_R(\text{addedLOC} - \text{deletedLOC})$
AVERAGE_CHURN	$\frac{\sum_R(\text{CHURN})}{ R }$
MAX_CHANGE_SET_SIZE	
AVERAGE_CHANGE_SET_SIZE	
AGE_IN_WEEKS	
WEIGHTED_AGE_IN_WEEKS	

- When Naive Bayes classifier is used, our results state that, in order to achieve better predictions accuracy, SMOTE technique performs better than any other. In fact, we were able to reach maximum value of Kappa and ROC parameter, while precision is very slightly less than result obtained with no sampling.

Kaufmann Publishers Inc., San Francisco, CA, USA.

- [7] T. Zimmermann, R. Premraj, and A. Zeller. 2007. Predicting Defects for Eclipse. In *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*. 9–9.

REFERENCES

- [1] Aalok Ahluwalia, Massimiliano Di Penta, and Davide Falessi. 2020. On the Need of Removing Last Releases of Data When Using or Validating Defect Prediction Models.
- [2] T. Gyimothy, R. Ferenc, and I. Siket. 2005. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering* 31, 10 (2005), 897–910.
- [3] Nachiappan Nagappan and Thomas Ball. 2005. Use of relative code churn measures to predict system defect density. *Proceedings - 27th International Conference on Software Engineering, ICSE05*, 284–292. <https://doi.org/10.1109/ICSE.2005.1553571>
- [4] Mrinal Rawat and Sanjay Dubey. 2012. Software Defect Prediction Models for Quality Improvement: A Literature Study. *International Journal of Computer Science Issues* 9 (09 2012), 288–296.
- [5] C. Tantithamthavorn, A. E. Hassan, and K. Matsumoto. 2018. The Impact of Class Rebalancing Techniques on the Performance and Interpretation of Defect Prediction Models. *IEEE Transactions on Software Engineering* (2018), 1–1.
- [6] Ian H. Witten, Eibe Frank, Mark A. Hall, and Christopher J. Pal. 2016. *Data Mining, Fourth Edition: Practical Machine Learning Tools and Techniques* (4th ed.). Morgan

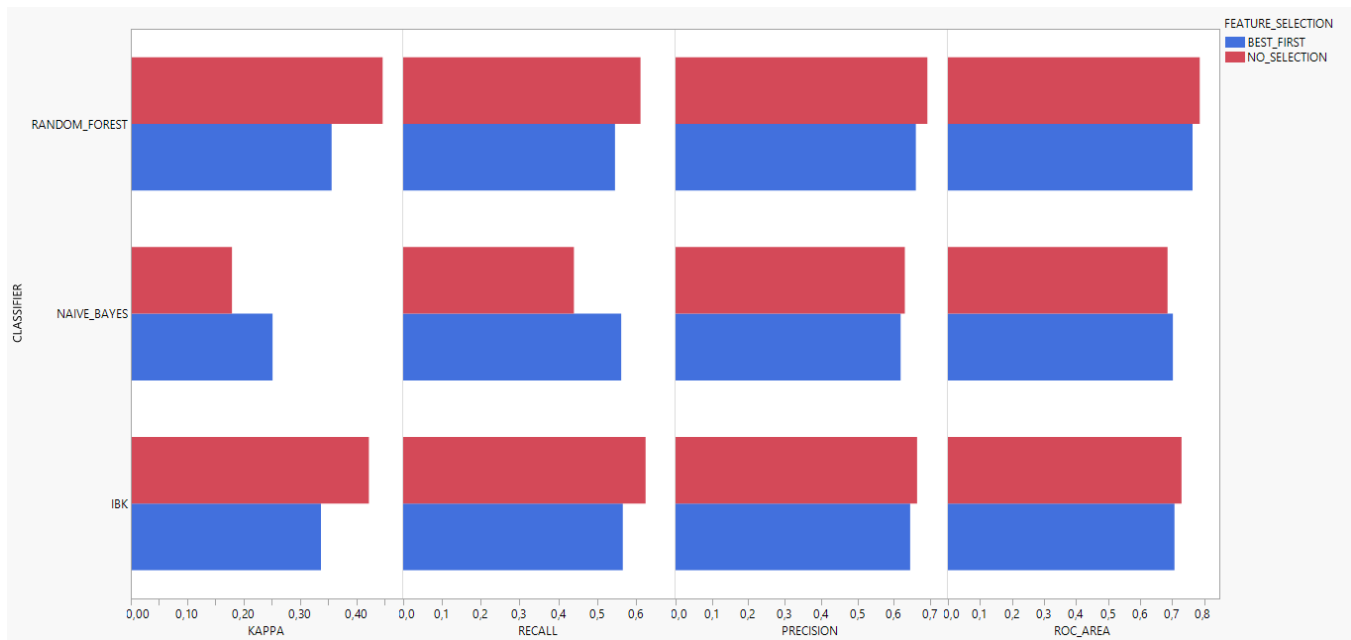


Figure 1: System Architecture

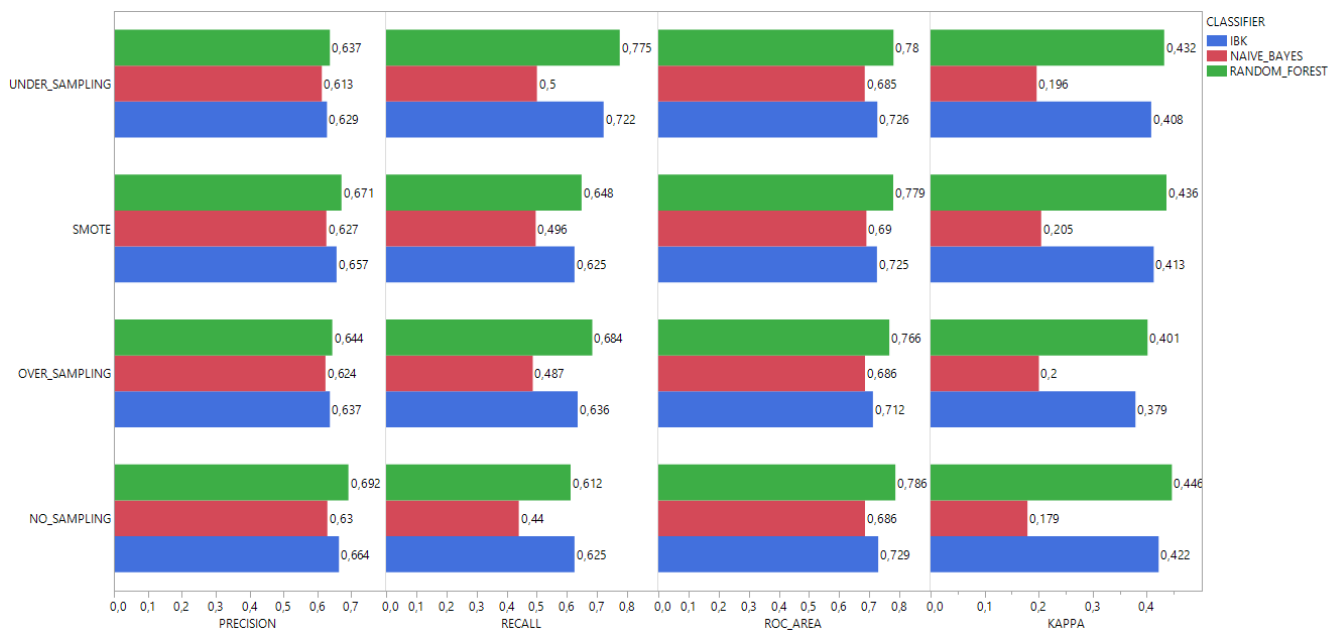


Figure 2: System Architecture