ISW2 Project A.A. 2019-2020

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

KEYWORDS

Software Defects Prediction, Feature Selection, Sampling, Walk-Forward

1 DELIVERABLE 2

1.1 Introduction

How to detect software modules which are more **defect-prone** than others? Why we are interested to identify them?

These are some of the most critical questions of software engineering, to which we will try to answer.

Firstly, 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].

As known, the presence of defective software modules *always incurs cost in terms of time and money*, 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?

In this report, we will describe a software defect prediction model based on **machine learning** techniques, which building passes through following steps:

- **Dataset building**
- **Dataset labelling**
- **Preprocessing**, using techniques like followings:
  - **Features Selection**
  - **Sampling**
- **Training and Testing**
- **Classification**

1.2 Dataset building

The first step is, obviously, to **generate a data-set**, in order to provide an appropriate input for our prediction model; in other words, we need to build a set of **instances**.

In our context, each data-set's instance represents a **source code file** belonging to one of the following open-source software projects which we have analysed:

Apache BookKeeper™ Which is a scalable, fault tolerant and low latency storage service[1].

Apache OpenJPA™ An implementation of the Java Persistence API specification[2].

Every data-set's instance is characterized by its values on a fixed, predefined set of **features**, or **attributes**, like its code lines number, its age, its complexity and so on. 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, scientific literature reports many studies, about software defect-proneness, which had 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.

1.2.1 *Our implementation.*

We extracted these defect-proneness correlated features from *history data* of our projects, exploiting tools like `git`[3], the most famous *version control systems* today available, and `JIRA`[4], which is a *issue tracking systems*.

From an implementation point of view, the `Git` class[5] contains the entire application logic to manage a `git` repository, implementing `VersionControlSystem` interface, reported in **listing 1**).

To be more precise, this class has following responsibilities:

- To identify the commit associated to each project's release.
- To extract metrics of a given source code file; this is done invoking `computeFileMetrics` method.
- To identify all source code files belonging to a release.

Listing 1: `VersionControlSystem` interface

```
public interface VersionControlSystem {

    Commit getCommitByTag(String tag);

    Commit getCommitByDate(LocalDateTime date);

    Commit getCommitByLogMessagePattern(String pattern);

    Map<String, File> getFiles(String commitHash);

    List<String> getFilesChangedByCommit(String commitHash);

    void computeFileMetrics(File file, Commit releaseCommit);
}
```

To be more precise, `computeFileMetrics` method was been developed in order to extract all metrics reported and described in **table 1**.

[1] https://github.com/apache/bookkeeper

[2] https://github.com/apache/openjpa

[3] https://git-scm.com/

[4] https://www.atlassian.com/it/software/jira

[5] Complete path: `datasetbuilder.datasources.vcs.git.Git`

Table 1: Metrics extracted from source code file X by `computeFileMetrics` method

Name	Description
LOC	Lines of code of X.
LOC_TOUCHED	$\sum_R(\texttt{addedLOC} + \texttt{deletedLOC})$ Where: • R is the set of all revisions of X. • addedLOC represents the number of added code lines. • deletedLOC represents the number of removed code lines.
LOC_ADDED	$\sum_R(\texttt{addedLOC})$
MAX_LOC_ADDED	$\max_R(\texttt{addedLOC})$.
AVERAGE_LOC_ADDED	$\frac{\sum_R(\texttt{addedLOC})}{\mid R \mid}$
NUMBER_OF_REVISIONS	$\mid R \mid$
NUMBER_OF_FIX	Number of bug fixes which have involved X.
NUMBER_OF_AUTHORS	The authors number which developed file X.
CHURN	$\sum_R(\texttt{addedLOC} - \texttt{deletedLOC})$
MAX_CHURN	$\max_R(\texttt{addedLOC} - \texttt{deletedLOC})$
AVERAGE_CHURN	$\frac{\sum_R(\texttt{CHURN})}{\mid R \mid}$
MAX_CHANGE_SET_SIZE	The number of files committed together with X.
AVERAGE_CHANGE_SET_SIZE	$\frac{\texttt{MAX\_CHANGE\_SET\_SIZE}}{\mid R \mid}$
AGE_IN_WEEKS	Age of X in weeks.
WEIGHTED_AGE_IN_WEEKS	$\frac{\texttt{AGE\_IN\_WEEKS}}{\texttt{LOC\_TOUCHED}}$

1.3 Dataset labelling

To be precise, our aim is to build a software defect predictor using a **supervised learning scheme**.

When a supervised learning scheme is used, the goal of our predictor 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 a *tag*, or a *label*, representing 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 *unlabelled data*.

Therefore, the second very important step to build our predictor is the so-called *labelling* task, according to which one or more informative tags must be assigned to each unlabelled instance of a given dataset.

1.3.1 *Our implementation.*

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

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

`Jira` class[6] contains application logic used to query `Jira` servers on the cloud. It implements the `IssueTrackingSystem` interface, shown in **listing 2**, providing the implementations for following methods:

`getReleases` Which is used to retrieve all necessary information about project's releases, including names, release dates and so on.

`getIssuesRegistry` Invoked to get all data about known issues

Listing 2: `IssueTrackingSystem` interface

```
public interface IssueTrackingSystem {

    List<Release> getReleases(String projectName);

    IssueRegistry getIssuesRegistry(
        String projectName,
        Map<Integer, Release> releasesByVersionID);
}
```

Querying `Jira` is not enough, because to proceed we need all commits associated to each release.

To do that, 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. The application logic to do that, is contained into `Git` class.

At this point, knowing commit associated to each release, we are able to retrieve all source code files associated to each project's release, however we still have to label them.

To do that we need to discover all known project's issues and, to do that, we 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 the implementation of this query procedure inside `getIssuesRegistry` method into `Jira` class.

Finally, to label each source code file as buggy or not, we need to check if the release, which the file belongs to, is an "*Affected Version*" (**AV**).

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 information about affected versions. 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

[6] Complete path: `datasetbuilder.datasources.its.jira.Jira`

is available in `Jira`, we considered the earliest AV in `Jira` to be the release that introduced the bug, otherwise we have adopted the so-called **proportion method**.

1.3.1.1 *The dormant defects problem.*

Empirical results show that is very common that a defect is discovered several releases *after* its introduction [1].

This fact implies that if we observe, today, the status of a source code file in its current version, it can be considered as defect-free while, in reality, this conclusion is *not* true because it can contain *undiscovered defects*.

In according to scientific literature, these undiscovered defects, commonly called "*dormant defects*" too, can negatively impacts our classifier's accuracy. We call "*snoring*" the "*noise*" consisting of all source code file containing dormant defects [1].

In order to remedy, literature suggest **to remove the most recent releases from a dataset**. In fact, since earlier releases likely contain more snoring classes than older releases, removing the most recent is possible to reduce the snoring effect improving our classifiers evaluations. Since scientific literature states that the first 50% of releases are not heavily impacted by snoring [1], for every analysed project, we have choose to remove from our dataset half of available releases. From an implementation point of view, this task is done executing `retrieveReleasesDiscardingHalfOfThem` method, which belongs to `ProjectDatasetBuilder` class [7].

1.4 Preprocessing

1.4.1 *Feature selection.*

After generating our data-set with metrics and labels, we applied a preprocessing technique called **feature selection** or **attribute selection**.

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, performances of machine learning systems degrades [6].

Therefore, using a feature selection technique, deleting unsuitable attributes, is possible to *improve learning algorithms performances*. However, using specific data-sets, is possible that they *may require extensive computations*, degrading overall performances [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.

For our analysis, we have adopted **Best-first** method. This scheme adopts a *greedy approach* to select an 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].

[7] Complete path: `datasetbuilder.ProjectDatasetBuilder`

At this point, our aim is to evaluate the advantages deriving from using a feature selection technique like *Best-First*. How to do that?

1.4.1.1 *How to evaluate classifier accuracy.*

According to project specifications, we have used following classifiers, which implementation is provided by `Weka`[8], an open source machine learning software:

- **Random Forest**
- **IBK**
- **Naive Bayes**

Our classifier's accuracy evaluation method consists of measuring following parameter:

Recall also known as **true positive rate**, measures correctly predicted buggy instances among all buggy instances:

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

Precision which tell us many times our classifier has correctly classified an instance as buggy:

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

AUC measures the area under the *receiver operating characteristic* (**ROC**) curve, plotted using *false positive rate* and *true positive rate* together.

Kappa tell to us how many times our classifier was been more accurate than a dummy classifier:

$$\frac{\text{Accuracy of classifier} - \text{Accuracy of dummy classifier}}{1 - \text{Accuracy of dummy classifier}} \tag{3}$$

1.4.1.2 *Evaluation results.*

Comparing results derived by processing our data-sets using best-first technique 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 observe from **fig. 1**, using our BookKeeper™ dataset, when a *Naive Bayes* classifier is used, exploiting *Best-First* method without any sampling scheme (which we will describe in next section), 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.

Our results confirm that *Naive Bayes* classifier improves its accuracy when *Best-First* method is used.

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

[8] `https://www.cs.waikato.ac.nz/ml/weka/`

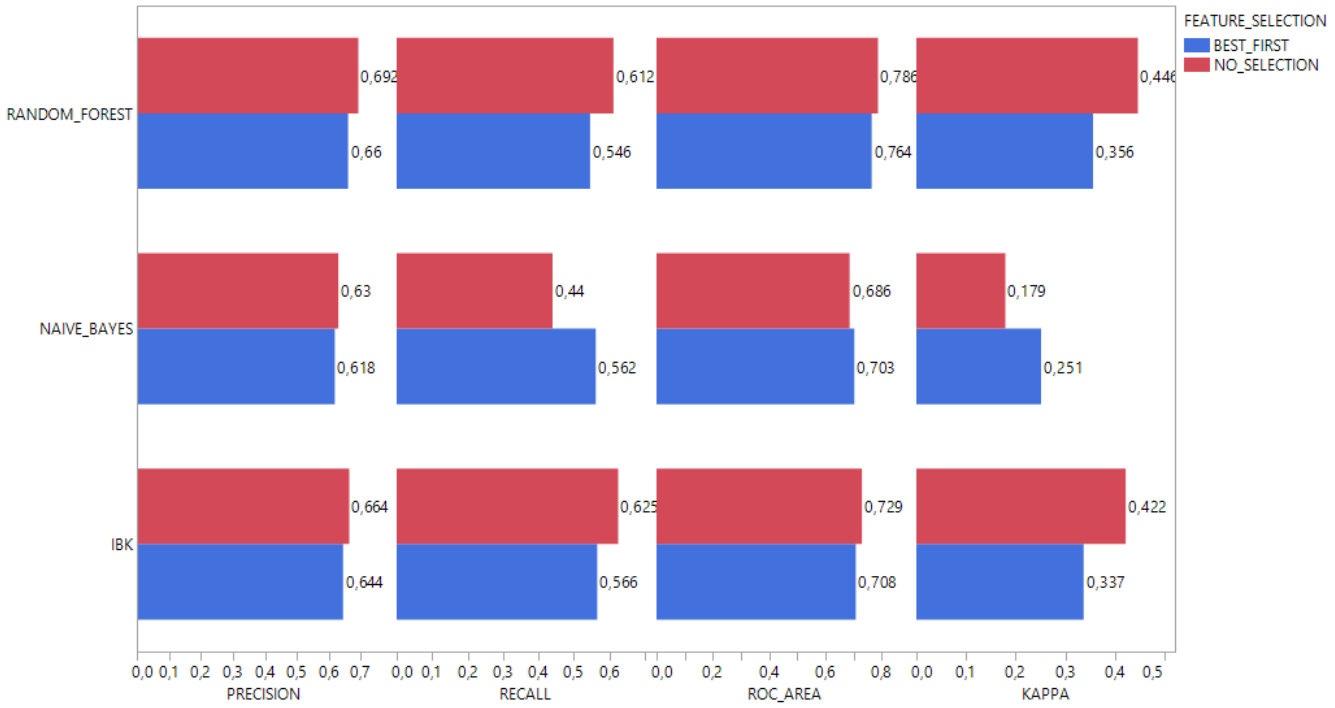

Figure 1: Comparison of classifiers accuracy when using a feature selection algorithm. No sampling technique applied. BookKeeper™ project.

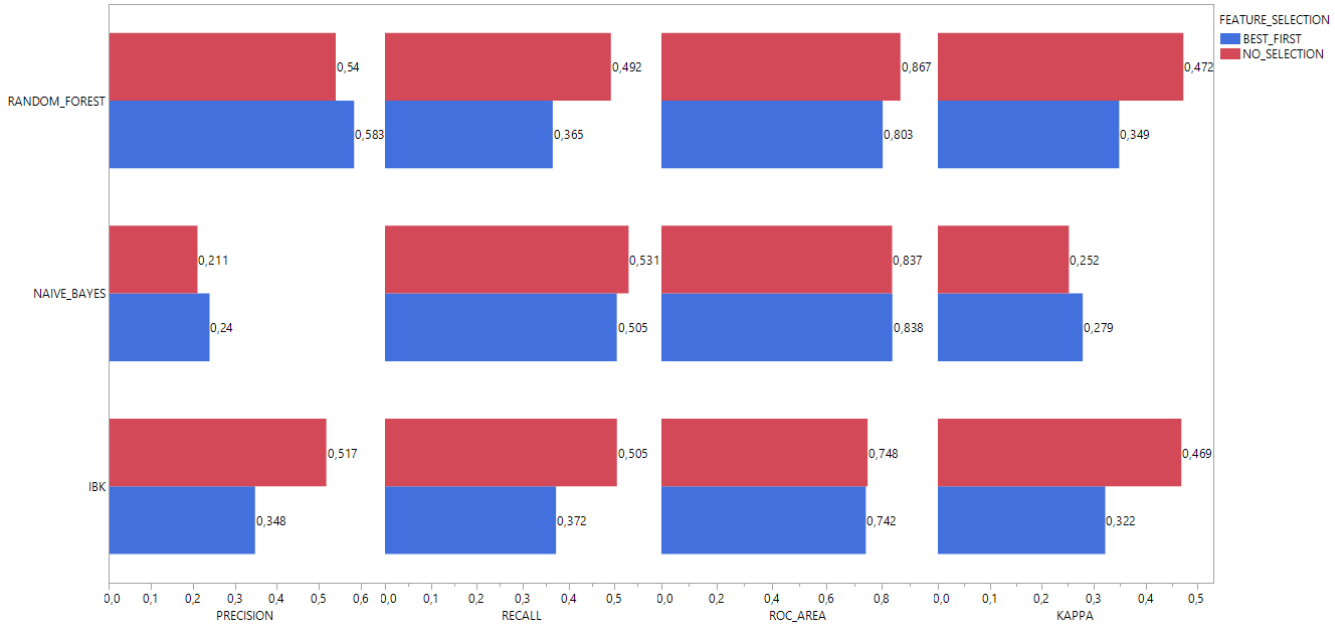

Figure 2: Comparison of classifiers accuracy when using a feature selection algorithm. No sampling technique applied. OpenJPA™ project.

- *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.

In other words, when *Random Forest* classifier is used, its accuracy get worse when *Best-First* method is applied. *IBK* classifier shows results very similar to those of *Random Forest*. Despite some differences, using OpenJPA™ dataset, results are similar.

1.4.2 Sampling.

As known, performance of a software defect prediction model depends heavily on the data on which it was trained.

In fact, when a defect prediction model is trained on **imbalanced datasets**, that is using a datasets where the proportion of defective and clean modules is not equally represented, it is *highly susceptible to producing inaccurate prediction models* [5].

To mitigate the risk of an imbalanced dataset, is possible to use several *sampling* techniques. In compliance with the project specifications, we have adopted following 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.

1.4.2.1 Our implementation.

The implementations of aforementioned sampling techniques are provided, once again, by `Weka` software.

To measure advantages deriving from using a sampling method, we evaluated our classifiers accuracy both when a sampling technique is exploited and when no sampling is used. In this case, no feature selection algorithm was been used.

- Using BookKeeper™ dataset,our results, shown in **fig. 3**, show that:
  - Regardless of the classifier used, *Recall* parameter improves always when any sampling technique is applied. In particular, *Random Forest* classifier using *under-sampling* achieves the highest *Recall* value, increasing by 26.63% the value achieved by the same classifier when no sampling is used.
  - Generally, *Precision* parameter gets little worse when any sampling technique is applied. The worst results are achieved using *Naive Bayes* classifier with *under-sampling* technique, decreasing by 2.77% the value achieved by the same classifier when no sampling is used.
  - *ROC* parameter gets very little worse when any sampling technique is applied too. The only exception is represented by *Naive Bayes* classifier with *SMOTE* technique which perform slightly better than when no sampling is used.
  - Regarding *Kappa* parameter, *Naive Bayes* performs very well when any sampling technique is applied, reaching best results when *SMOTE* technique is used. Other classifier perform generally worse. The worst results are achieved using *IBK* classifier with *over-sampling* technique.
- Using OpenJPA™ dataset, our results, shown in **fig. 4**, show that:
  - Like previously, regardless of the classifier used, *Recall* parameter improves always. In particular, *Random Forest* classifier using *under-sampling* achieves the highest *Recall* value, increasing by 64.63% the value achieved by the same classifier when no sampling is used; a very good results.
  - Generally, *Precision* parameter gets little worse when any sampling technique is applied. All classifier perform very bad when *under-sampling* is used. The only exception is represented by *Naive Bayes* classifier with *SMOTE* technique which perform slightly better than when no sampling is used.
  - *ROC* parameter gets very little worse when any sampling technique is applied too. The only exception is represented by *Random Forest* classifier with *SMOTE* technique which performs better.
  - Using OpenJPA™ dataset, regarding *Kappa* parameter, all classifier perform very bad when *under-sampling* is used. *Naive Bayes* performs better when *SMOTE* is used. In all other case, worse results are achieved.

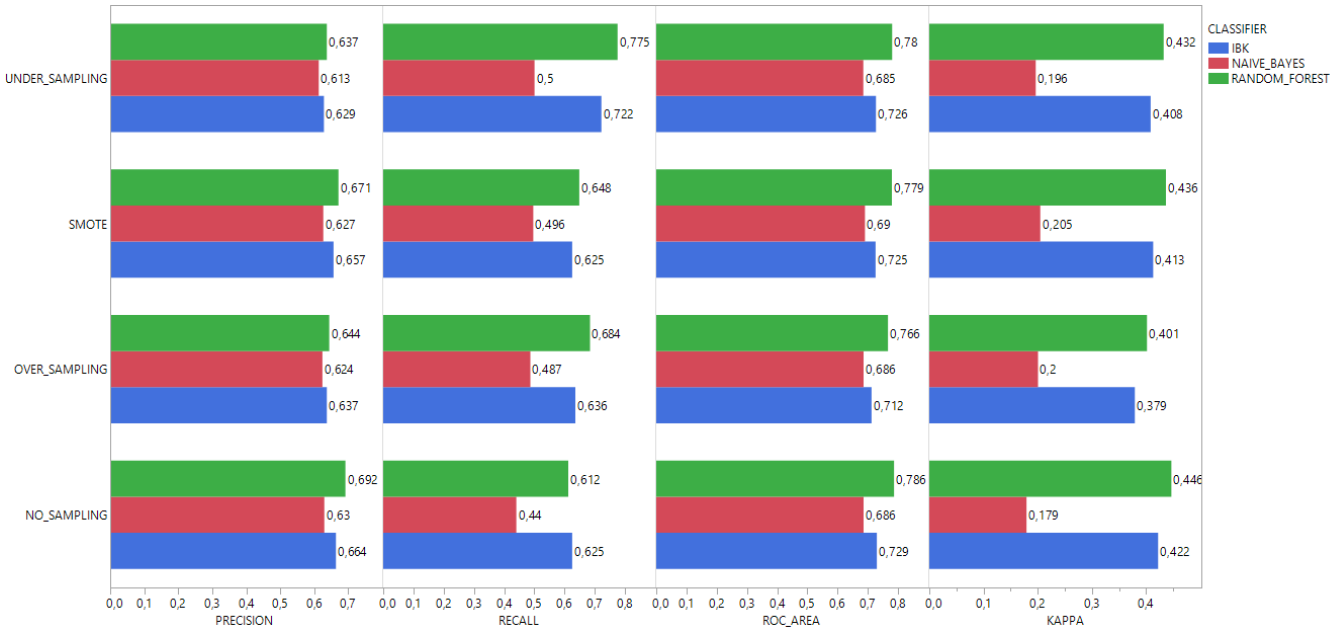

Figure 3: Comparison of classifiers accuracy when using sampling algorithms. No feature selection technique applied. BookKeeper™ project.

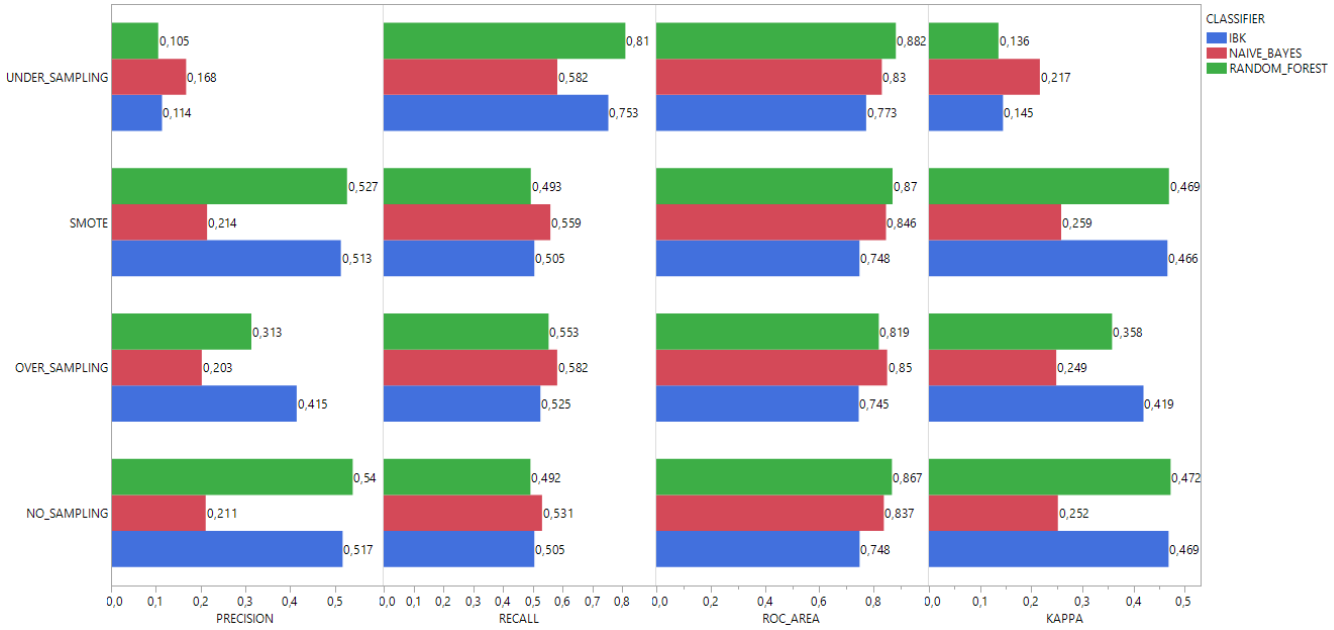

Figure 4: Comparison of classifiers accuracy when using sampling algorithms. No feature selection technique applied. OpenJPA™ project.

1.5 Training and Testing

At this point we can train our software defect prediction model but, firstly, we will talk about **validation techniques**. Why are they so important?

Validation techniques describe a specific way to split available data in *training set* and *test set* and they are extremely important to do our evaluations.

First of all, is very important to emphasize that the data-set involved during training phase *cannot* be used to measure above parameters due to following reasons:

- We *already* know the classifications of each instance belonging to training set, therefore we are not interested in accuracy about predictions involving these data.
- Any estimate of performance based on training data will be, clearly, optimistic and, therefore, not very useful.

Therefore, to evaluate classifier's performance, we have to use a new data-set, containing instances *not* used during training phase; this independent dataset is the test set. To identify our test set we need to use a validation technique.

1.5.1 *The walk-forward validation technique.*

In accordance to project's specifications, we have adopted a very common validation technique called **Walk-forward**. Precisely, it is a *time-series* validation technique because it has the ability to *preserve the temporal order of data*, preventing that testing set have data antecedent to the training set.

According to walk-forward technique, a data-set must be split into parts, which must be the smallest units that can be *chronologically* ordered. Therefore, during the development of our project, our data-set was been divided into parts containing all source code files belonging to each project's release, the latter are been chronologically ordered respect to their release date.

Then and in each run, all data available before the part to predict is used as the training set, and the part to predict is used as test-set. Afterwards, the model accuracy is computed as the average among runs. The number of runs is equal to one less than the number of parts.

For example, regarding **BookKeeper™** project, we have built a dataset containing 6 releases, therefore all data available were divided into 6 parts, while the number of runs is equal to 5.

Particularly, during first run, the first part is used as a training, and the second as testing. During the second run the first two parts are used as a training and the third as testing, and so on.

1.5.1.1 *Some experiments.*

At this point, we wanted to know how accuracy's parameters behave according to the number of releases used for training using walk-forward validation technique. Results, reported in **fig. 5** and **fig. 6**, show that:

- Regarding BookKeeper™ project, regardless of the classifier used, all parameters improve linearly, increasing the number of releases used for training. Better results are achieved using 6 releases during training phase.
- Regarding OpenJPA™ project, the graph shows a very irregular trend about all parameters. However, regardless of the classifier used, better results are achieved using 9 releases, however they start to get worse immediately after.

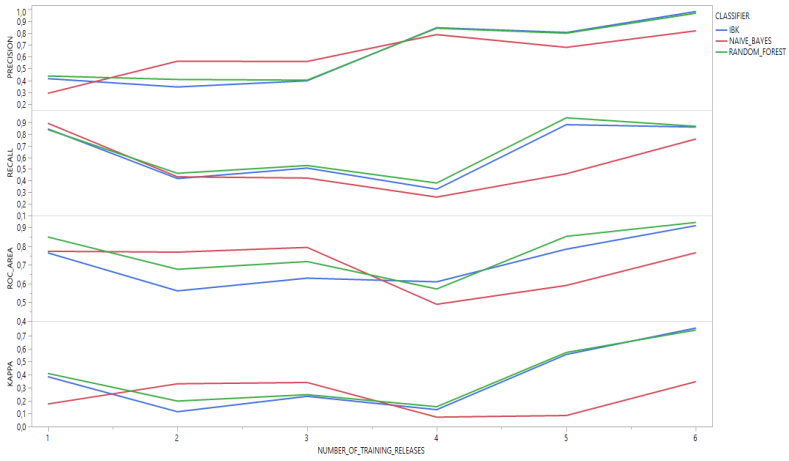

Figure 5: Trend of the accuracy's parameters respect to the releases number.
BookKeeper™ project.

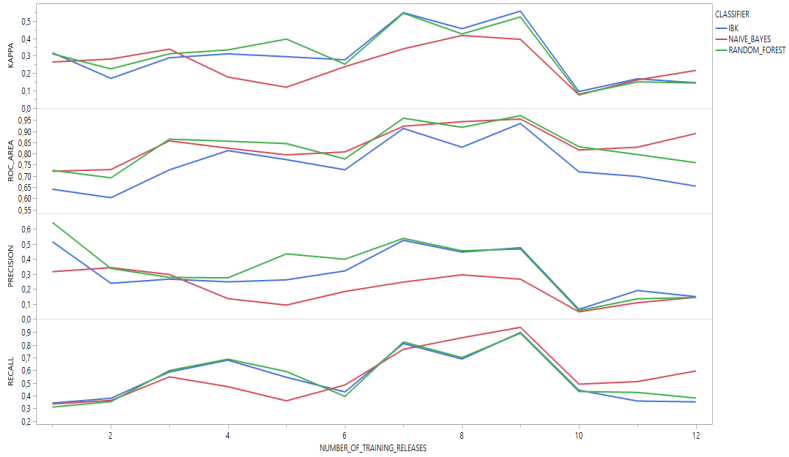

Figure 6: Trend of the accuracy's parameters respect to the releases number.
OpenJPA™ project.

1.5.1.2 *Final comparison.*

From accuracy point of view, what is the best classifier? What sampling or feature selection algorithm perform better with our datasets?

To reply to these question, we have to simply observer **fig. 7** and **fig. 8**, which contain a summary evaluation of all possible combination of algorithms and classifiers using BookKeeper™ and OpenJPA™ datasets respectively.

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

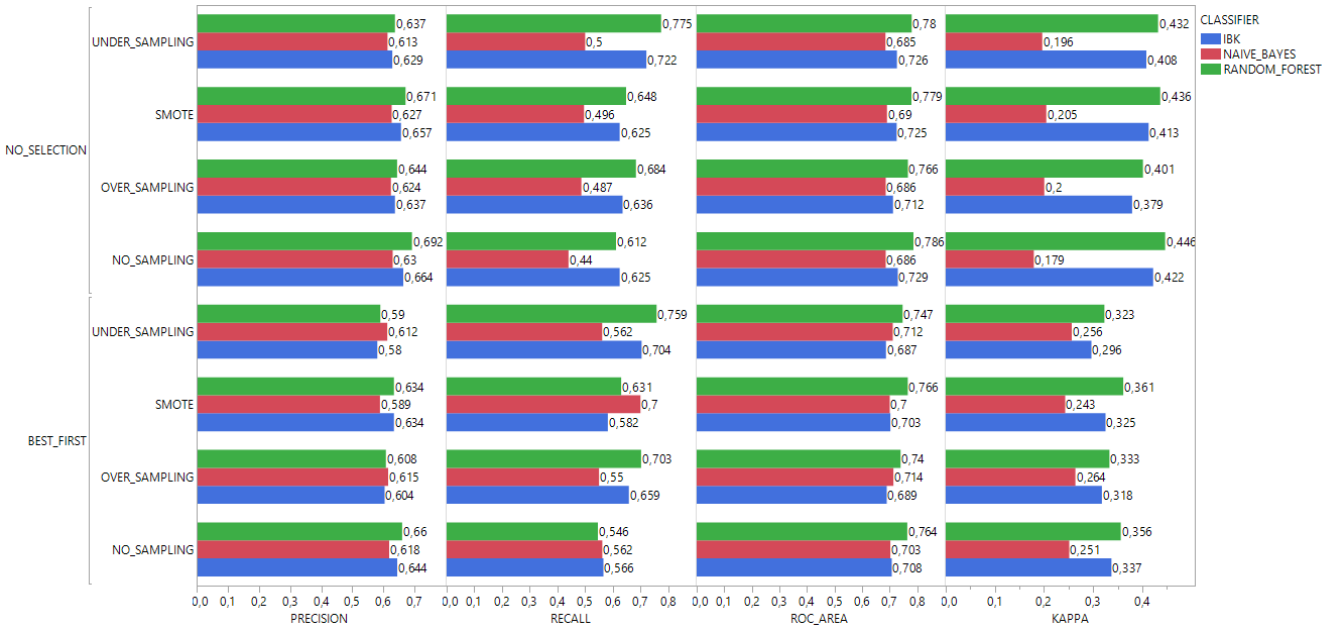

Figure 7: Summary of accuracy of our classifiers. BookKeeper™ project.

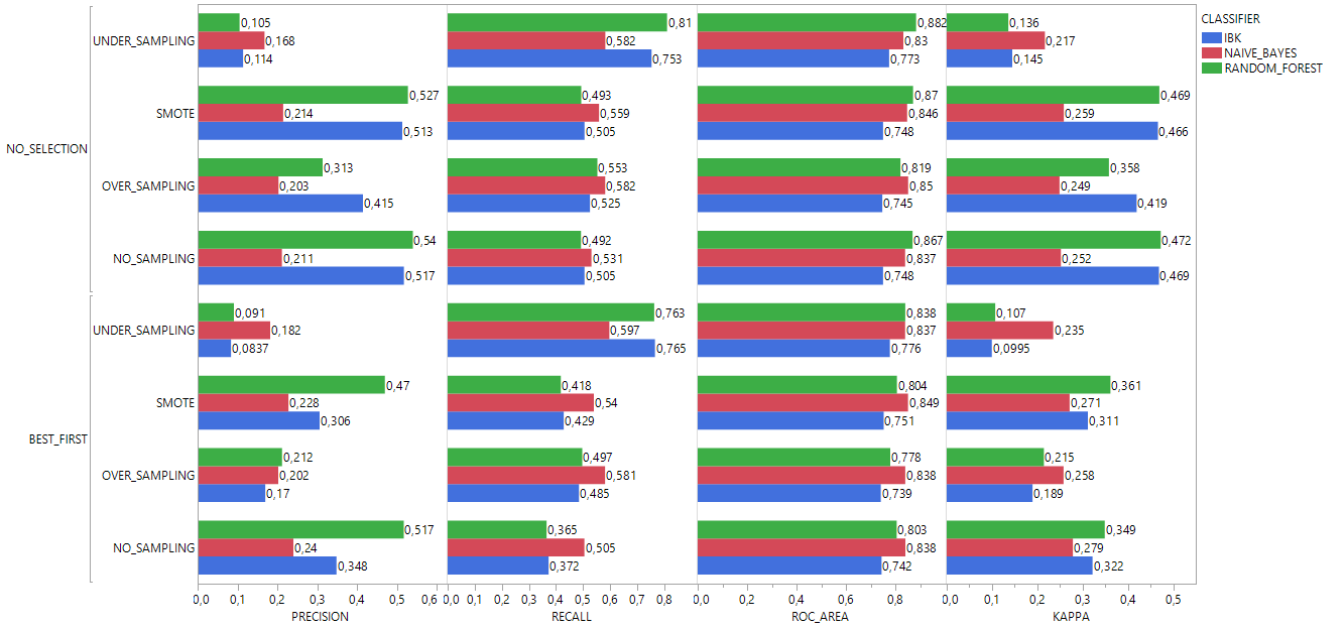

Figure 8: Summary of accuracy of our classifiers. OpenJPA™ project.

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 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.