Understanding Software Defect Prediction Models

Shrenuj Gandhi
Computer Science
North Carolina State University
sgandhi4@ncsu.edu

Neng Jiang
Computer Science
North Carolina State University
njiang@ncsu.edu

***Abstract*— Software defect prediction is one of the most assisting activities of the testing phase of software development lifecycle. It identifies modules that are defect prone and require extensive software testing. This way the testing resources can be used efficiently without violating the constraints.**

In this paper, we analyze a baseline study and replicate its results. We apply sampling technique after analyzing the datasets and enhance the results by overcoming the curse of dimensionality. Finally, we take this setup and apply it to different set of learners to gain insight into their performances.

***Index Terms*— Halstead metrics, McCabe metrics, naive Bayes, Information Gain, cross-validation, SMOTE, Logistic Regression**

I. INTRODUCTION

Software defect prediction strives to improve software quality and testing efficiency by constructing predictive classification models from code attributes to enable a timely identification of fault-prone modules.

Risk Analysis and Management are a series of steps that help a software team to understand and manage uncertainty. Many problems can plague a software project. A risk is a potential problem it might happen, it might not. But, regardless of the outcome, its a really good idea to identify it, assess its probability of occurrence, estimate its impact, and establish a contingency plan should the problem actually occur. Everyone involved in the software process managers, software engineers, and customers participate in risk analysis and management.

To overcome risk, Software Defect Prediction can help to detect defects and identify potential defects. There are different fault prediction approaches available in the Software Engineering discipline. Such as fault prediction, security prediction, effort prediction, reusability prediction, test effort prediction and quality prediction. These approaches help us to minimize the cost of testing which minimize the cost of the project. Many research studies in a decade have focused on proposing new metrics to build prediction models. Widely studied metrics are Source Code and Process Metrics. 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 that manage all development histories. Process Metrics quantify many aspects of software development process such as changes of source code, ownership of source code files, developer interactions, etc. Usefulness of process metrics for defect prediction has been proved in many studies. Most defect prediction studies are conducted based on statistical approach, i.e. machine learning. 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).

In this paper, we are going to focus on Halstead and McCabe metrics which fall under the category of source code metrics. Due to noisy data, it is very difficult to predict the exact number of defects in a software module. Thus we stick to general classification problem.

The rest of the paper is organized as follows. Section II discusses the baseline study. It talks about the methodology used and the results achieved using statistical based learning method. Sections III is based on baseline study with every sub-section producing some interesting results. Sub-section A mentions steps taken to replicate the baseline study in Python. Sub-section B describes why and how Synthetic Minority Oversampling Technique (SMOTE) is applied. Using SMOTE, we tune the feature extraction step in sub-section C. We put different learning methods to compare their predictive capabilities in sub-section D. In Section IV, we reason why our approach is software science centric. Finally, in Section V and VI, we conclude with our results from the experiments and how it can be extended in the future.

II. BASELINE STUDY

In the paper, Data Mining Static Code Attributes to Learn Defect Predictors, authors Tim Menzies, Jeremy Greenwald and Art Frank demonstrated that defect predictors are demostrably useful [1]. They compared different learner and found that *nave Bayes* data miner with a *log filtering* preprocessor on the numeric data produced better results with a mean probability of detection of 71% and mean false alarm rates of 25%.

Studies prior to this paper had reached different conclusions because they were based on different data. Since this paper used public-domain data sets [2], it allowed different researchers to use and compare their techniques. This is one of the reason of making this study our baseline paper.

```
M    = 10
N    = 10
All  = 38  # all the attributes
DATAS=(cm1 kc3 kc4 mw1 pc1 pc2 pc3 pc4) # data set list
FILTERS= (none logNums)                 # filter list
LEARNERS= (oneR j48 nb)                 # learner list

for data in DATAS
  for filter in FILTERS
     data' = filter(data)
     rank data' attributes via InfoGain # Equation 2
      for i = 1,2,3, All
        attribute' = the i-th highest ranked attributes
        data''     = select attributes' from data'
        repeat M times
           randomized order from data''
           generate N bins from data''
           for i in 1 to N
             tests        = bin[i]
             trainingData = data'' - tests
             for learner in LEARNERS
               METHOD         = (filter attributes' learner)
               predictor      = learner(trainingData)
               RESULT[METHOD] = apply predictor to tests
```

Fig. 1. Data is filtered and the attributes are ranked using InfoGain. The data is then shuffled into a random order and divided into 10 bins. A learner is then applied to a training set built from nine of the bins. The learned predictor is tested on the remaining bin.

The learning problem is to build some predictors which guesses the labels for as-yet-unseen modules. Using the method described in Figure 1, this paper offers new defect predictors with a probability of prediction and probability of false alarm of 71% and 25%.

The authors encountered an interesting pattern in the datasets. All the numeric data attributes had an exponential distribution with many small values and a few larger values. Previous experiments proved that logarithmic filter on all numeric values improves predictor performance [4]. Hence the *log filter* step replaces all the numeric values *n* with their logarithms *ln(n)*.

The study uses (M = 10) * (N = 10) - way cross-evaluation iterative attribute subset selection shown in Figure 1. The data set is divided into N buckets. For each bucket in a 10-way cross-evaluation, a predictor is learned on nine of the buckets, then tested on the remaining bucket. This step wraps inside scripts that explores different subsets of the attributes in the order suggested by InfoGain. In the innermost loop of the study, some method was applied to some data set. As shown in the third to the last line of Figure 1, these methods were some combination of filter, attributes, and learner.

InfoGain is a measure of purity. It represents the expected amount of information that would be needed to specify whether a new instance should be classified defective or not, given the example that reached the node.

III. METHODOLOGY

As part of this project, we perform 4 experiments. The data and its format used in the experiments is discussed in Appendix A. The experiments load the data from the files

```
DATASETS = read file from database.txt
M = 10
N = 10
LEARNERS = ['NB']
METHODS = {'NB':naiveBayes}

for dataset in DATASETS:

    table = Table (dataset)

    # LOG FILTERING
    table = log_filtering(table)

    # FEATURE SELECTION
    for i,col in enumerate(columns):
        entropyDict[col] = calculate InfoGain for each column

    top_ranked_attributes = dict (sorted (entropyDict.items(), key=operator.
        itemgetter(1))[:3])
    table.rows = keep only top_ranked_attributes and class_varaible

    for index in range(M):

        shuffle (table.rows)

        X = np.array ([x[:-1] for x in table.rows])
        y = np.array ([x[-1] for x in table.rows])
        skf = StratifiedKFold (n_splits=N, random_state=None, shuffle=True)

        for train_index, test_index in skf.split(X, y):
            train, test = X[train_index], X[test_index]
            train_label, test_label = y[train_index], y[test_index]

            for learner in LEARNERS:
                # RUN LEARNER
                predicted = METHODS[learner] (train, train_label, test)
```

Fig. 2. Algorithm used to implement baseline study

mentioned in *database.txt* and with the assumption that the data is in a csv format. During processing of data, we assume that all columns are numeric and for evaluation purposes we use recall and false alarm measures displayed by the standard Snott-Knot test.

A. Reproducing Baseline Study

The baseline study was performed in WEKA. As software scientist we started with the task of reproducing the study in Python. Figure 2 shows the algorithm used to implement the script in Python.

The cross-validation step is done using stratified sampling. Stratified k fold cross validator breaks the data into training and testing sets while preserving the percentage of samples for each class. Many algorithms exhibit *order effects*, where certain ordering dramatically improves or degrades performance. To overcome order effects we shuffle the records randomly, each time we are about to generate testing and training sets. The learner used in this experiment is skicit-learn's Gaussian Naive Bayes classifier with default parameters [5][6]. And the logic for logarithmic filter is as follows:

```
min_val = 0.000001
  for each value in table
    if (value <= min_val):
          value = log(min_val)
       else:
          value = log(value)
return table
```

With the above mentioned procedure we get, Figure 3, a mean recall of 59% (against 70% in baseline study) and a mean false alarm 18% (against 24% in baseline study).

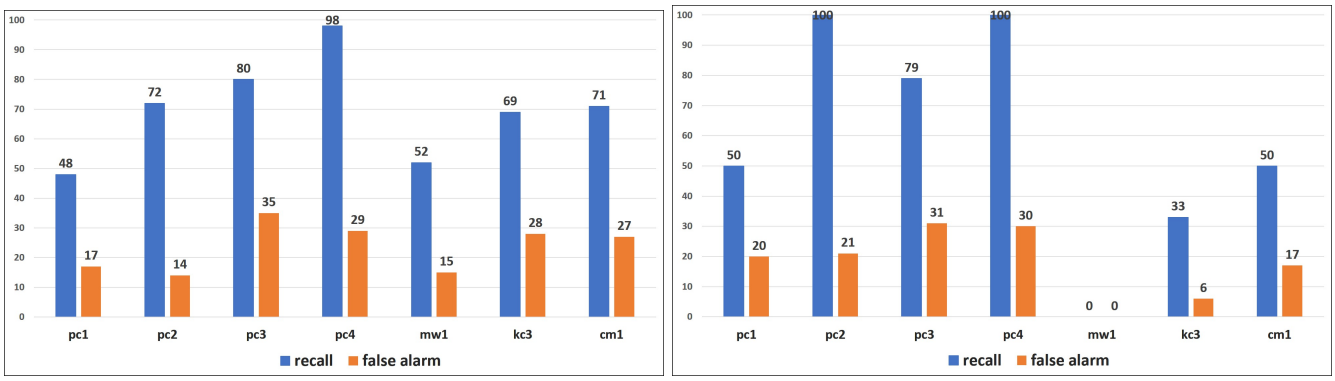

Fig. 3. The left hand side graph are the baseline study results. It shows the average recall and average false alarm rate of Naive Bayes classifier with log filtering when top 3 features (based on InfoGain) are selected. The right hand side graph depicts the results we produced by following the steps in stated in Figure 2.

The results imply that python implementation produces completely different results for most of the datasets. Possible reasons for this behavior could be difference in parameters of naive Bayes classifier, and difference in implementation of how InfoGain is calculated. Since the results were poor, as compared to baseline study, we look at a technique to optimize results.

B. Performing SMOTE

The efficiency of software fault prediction models is greatly shaped by the class distribution of the training data. Class distribution is described as the number of instances of each class in the training dataset. If the number of instances belonging to one class is much more than the number of instances belonging to another class, then the problem is known as class imbalance problem. Figure 4 shows the percentage of defective class in our datasets. It implies that the dataset clearly suffers from class imbalance problem. The class with more instances is called majority class and the one with lesser instances is called minority class. The problem widens when the class under consideration, i.e. the faulty class is represented by fewer instances. Various techniques have been proposed for addressing this problem - including re-sampling, SMOTE, and penalized models [7].

In this experiment we apply the Synthetic Minority Over-Sampling Technique (SMOTE). It works by creating synthetic samples from the minor class instead of creating copies. The algorithm selects two or more similar instances (using a distance measure) and perturbing an instance one attribute at a time by a random amount within the difference to the neighboring instances [8].

In python, we use skicit learn's imbalanced-learn library. It offers various oversampling methods like SVM, regular, random and ADASYN. For this experiment, we use the regular algorithm, which is best explained in Figure 5, with default parameters set as

```
SMOTE (k = 5, kind = 'regular', m = 10,
    n_jobs = -1, out_step = 0.5,
    random_state = None, ratio = 'auto')
```

dataset	defective = true	% defective
pc1	61	8.65
pc2	16	2.14
pc3	134	12.44
pc4	178	12.20
mw1	27	10.67
kc3	36	18.55
cm1	42	12.84

Fig. 4. class distribution in overall datasets. Since none of the datasets have more than 20% defective class variables, our learner will not have enough records to learn from and thus would not be able to predict them correctly.

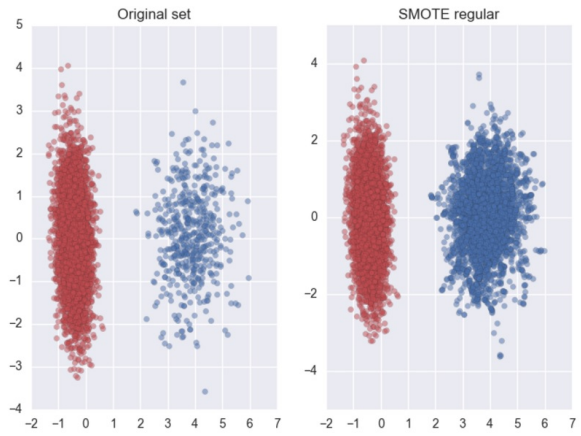

Fig. 5. How SMOTE regular over-samples the minority class

After applying SMOTE, we there is an increase in the learner's performance. Figure 6 depicts the increase in recall values after SMOTE is applied to the training set. The mean recall rises from 72% to 86%, when SMOTE is applied and top 5 ranked attributes are selected.

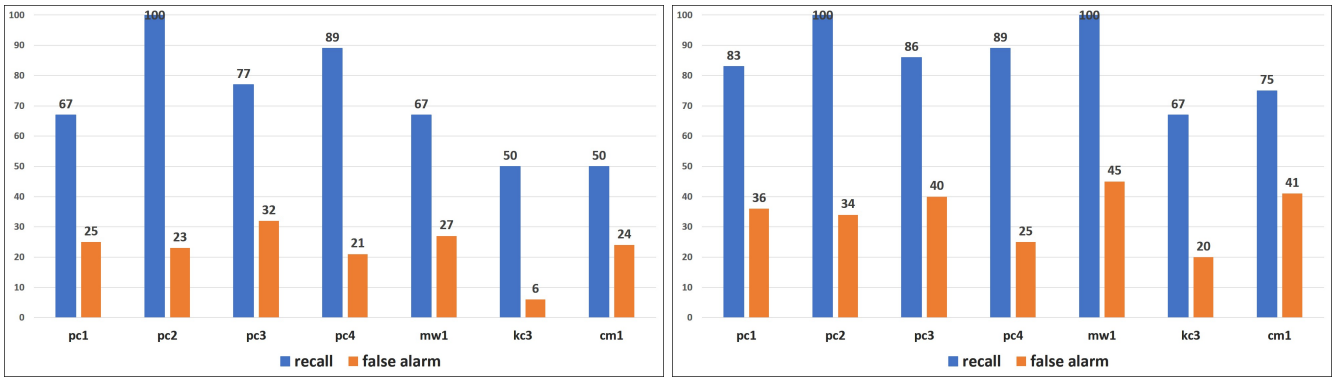

Fig. 6. The left hand side graph shows average recall and average false alarm values calculated by Naive Bayes after log filtering when top 5 attributes are selected based on InfoGain. The right hand side graph shows the average recall and average false alarm values when SMOTE is applied to the training set, while keeping the other parameters same.

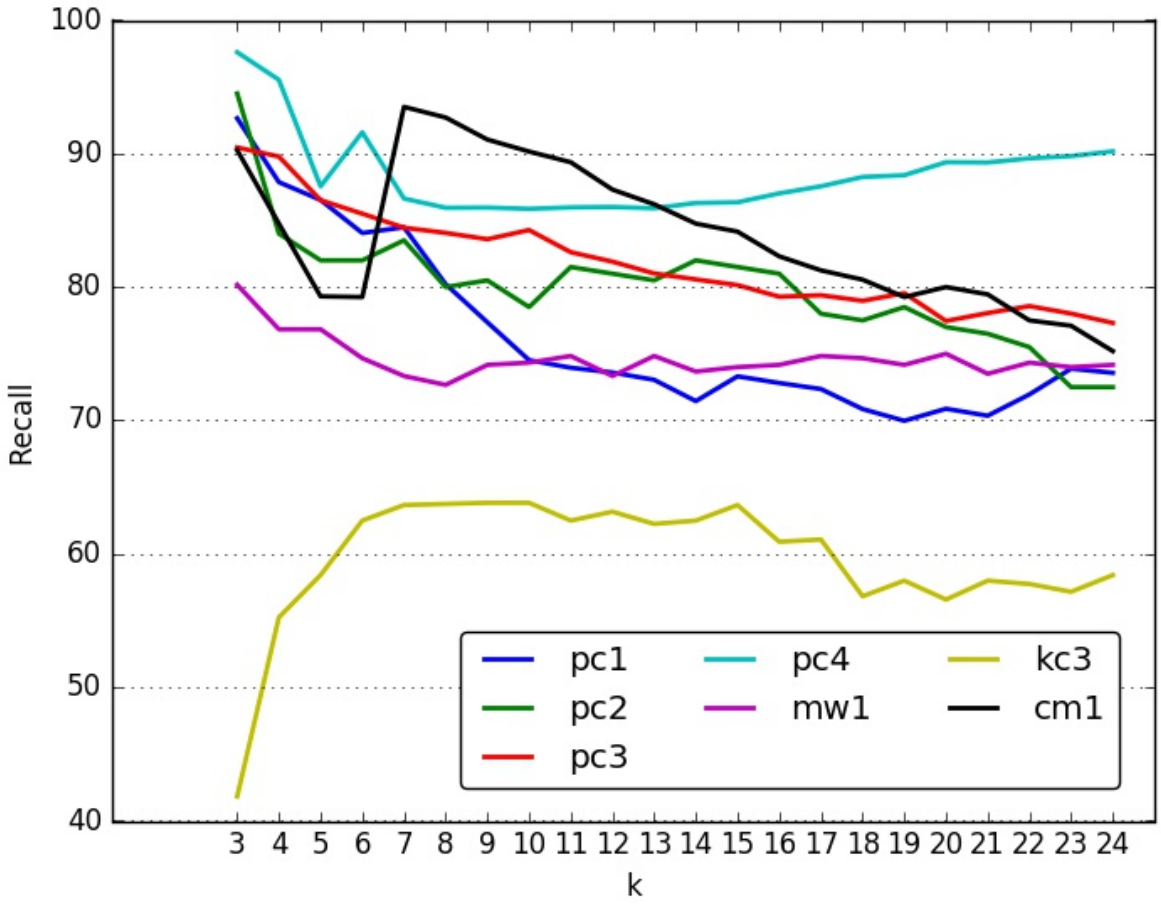

Fig. 7. Recall values for different datasets, when top k attributes are selected. The procedure performs SMOTE on training set and uses naive Bayes with log filtering.

Datasets	Experiment – Tuning Feature Selection				Baseline Study			
	Recall	*False Alarm*	*Value of k*	*Attributes Selected*	*Recall*	*False Alarm*	*Value of k*	*Attributes Selected*
pc1	92.5	36.6	3	NUM_UNIQUE_OPERANDS, LOC_BLANK, HALSTEAD_CONTENT	48	17	3	CALL_PAIRS, NUM_UNIQUE_OPERANDS, NUMBER_OF_LINES
pc2	91.5	39.0	3	HALSTEAD_CONTENT, PERCENT_COMMENTS, LOC_CODE_AND_COMMENT	72	14	2	LOC_COMMENTS, PERCENT_COMMENTS
pc3	90.4	47.4	3	PERCENT_COMMENTS, LOC_BLANK, NUM_OPERANDS	80	35	3	LOC_BLANKS, HALSTEAD_CONTENT, NUMBER_OF_LINES,
pc4	97.5	30.4	3	LOC_CODE_AND_COMMENT, PERCENT_COMMENTS, LOC_TOTAL	98	29	3	LOC_BLANKS, LOC_CODE_AND_COMMENT, PERCENT_COMMENTS,
mw1	78.5	48.5	3	LOC_COMMENTS, NUMBER_OF_LINES, DESIGN_DENSITY	52	15	3	HALSTEAD_ERROR_EST, NODE_COUNT, NUM_UNIQUE_OPERANDS
kc3	63.5	33.0	10	LOC_COMMENTS, DESIGN_DENSITY, LOC_CODE_AND_COMMENT, CYCLOMATIC_COMPLEXITY, HALSTEAD_VOLUME, HALSTEAD_CONTENT HALSTEAD_ERROR_EST, CALL_PAIRS, ESSENTIAL_DENSITY, GLOBAL_DATA_COMPLEXITY	69	28	3	LOC_EXECUTABLE, HALSTEAD_LEVEL HALSTEAD_PROG_TIME
cm1	93.6	51.9	7	HALSTEAD_VOLUME, LOC_EXECUTABLE, LOC_COMMENTS, NUM_UNIQUE_OPERANDS, NUM_OPERATORS, CYCLOMATIC_DENSITY, PERCENT_COMMENTS	71	27	3	CALL_PAIRS, NUM_UNIQUE_OPERANDS, NUM_UNIQUE_OPERATORS

Fig. 8. Comparison between attributes selected in this experiment vs attributes selected in baseline study

C. Tuning Feature Selection

There are lots of different software metrics discovered and used for defect prediction in the literature. Instead of dealing with so many metrics, it would be practical and easy if we could determine the set of metrics that are most important and focus on them more to predict defectiveness[9].

In the baseline study, we saw that based on iterative and exhaustive subsetting, top 3 attributes were chosen. In this sub-section, we discuss how we tuned our selection process. We run a loop starting from 3 to 25 attributes and calculate the recall. We select top k attributes that give us the best recall measures. Figure 7 plots the recall values for different datasets with value of k ranging from 3 to 25. We can see that, except for kc3 and cm1, we get better performances when value of k = 3. Interestingly, the baseline paper also selects top 3 attributes while producing its results. Figure 8 lists the attributes selected by tuning experiment vs the attributes selected in the baseline study. After tuning, we get a mean recall of 86.7% (against 70% in baseline study) and a mean false alarm rate of 40.9%(against 24% in baseline study).

D. Comparing Learners

To get a good grasp of the performance of different learners, we used, in this experiment, Support Vector Machine (SVM), Random Forest, Classification and Regression Trees(CART), and Logistic Regression(LR).

SVM method uses points in a transformed problem space that best separate classes into two groups. Classification for multiple classes is supported by a one-vs-all method. SVM also supports regression by modeling the function with a minimum amount of allowable error. Random Forest is an extension of Bootstrap Aggregation or bagging that in addition to building trees based on multiple samples of your training data, it also constrains the features that can be used to build the trees, forcing trees to be different. CART are constructed from a dataset by making splits that best separate the data for the classes or predictions being made. Logistic regression fits a logistic model to data and makes predictions about the probability of an event (between 0 and 1) [15].

Figure 9 shows performance of different learners for dataset pc3. We got similar results for the the rest of he datasets where naive bayes classifier outperforms other learners.

```
pc3.csv

RECALL FOR :  pc3.csv

rank ,         name ,    med   ,  iqr
----------------------------------------------------
   1 ,         CART ,    0.43  ,  0.19 (       ----*   |             ), 0.31,  0.43,  0.50
   2 ,           RF ,    0.50  ,  0.21 (         ----* |             ), 0.36,  0.50,  0.57
   3 ,          KNN ,    0.64  ,  0.23 (             -|--*           ), 0.54,  0.64,  0.77
   4 ,          SVM ,    0.85  ,  0.15 (              |      --*     ), 0.77,  0.85,  0.92
   4 ,           LR ,    0.85  ,  0.15 (              |      --*     ), 0.77,  0.85,  0.92
   5 ,           NB ,    0.92  ,  0.08 (              |        ---*  ), 0.85,  0.92,  0.93

FALSE ALARM :  pc3.csv

rank ,         name ,    med   ,  iqr
----------------------------------------------------
   1 ,           RF ,    0.13  ,  0.05 ( --*          |             ), 0.11,  0.14,  0.16
   1 ,         CART ,    0.15  ,  0.06 (  --*         |             ), 0.12,  0.15,  0.18
   2 ,          KNN ,    0.23  ,  0.05 (       --*    |             ), 0.21,  0.23,  0.27
   3 ,          SVM ,    0.31  ,  0.06 (           --* |             ), 0.28,  0.31,  0.34
   4 ,           LR ,    0.35  ,  0.06 (            --*             ), 0.32,  0.35,  0.38
   5 ,           NB ,    0.47  ,  0.07 (              |    --*       ), 0.44,  0.48,  0.51

RUNTIME
'KNN': 0.5069994926452637, 'SVM': 12.60900092124939, 'NB': 0.08899903297424316, 'CART': 0.9129993915557861, 'RF':
6.581000328063965, 'LR': 0.5560007095336914
```

Fig. 9. Performance of different learners for pc3 dataset

IV. Software Science vs Data Dabbling

The fundamental objective of this paper was to reproduce the baseline study results. While most of data mining is done through hypothesis testing, and limited data set, it is a good idea to check these results once in a while. It is all the more useful to test these, if they were performed using different tools.

Our results are not golden. We invite fellow researchers to test our approach and notify us by raising issues in our GitHub repository[3]. The steps for all the above experiments, along with sample outputs and data sets is present on our repository.

V. Conclusion

From experiment 1, we conclude that, replicating studies can be complicated and such studies should contain link to scripts or steps to replicate them.

From experiment 2, we can say that software defect prediction is often faced with imbalanced class problem. There are a list of ways we can handle this problem [7] and SMOTE'ing the training set is a good solution.

Defect prediction is data dependent. Thus if 3 attributes almost describe the distribution of class variable in one dataset, does not mean it will apply to different datasets. It is also interesting to note that not all attributes carry value. Via feature selection in experiment 3, we establish that only a small subset of attribute contribute to the answer of whether the module is defective or not. The size of the subset depends on the dataset. According to Figure 7, kc3 requires the most attributes. A likely reason could be the language. Maybe Halstead and McCabe and metrics work best for C-based modules. But this is just speculation. More research is required in this aspect to conclude this behavior.

Our conclusion is that data mining static code attributes to defect predictor is useful. Along the lines of the baseline paper, we conclude that how the attributes are used to build predictors is much more important than which particular attributes are used.

VI. Future Work

In future, we can test other solutions like penalized models and re-sampling techniques to handle the imbalanced class problem and compare the results with SMOTE. Within SMOTE, we can tune its parameters to get better results. This also applies to the different learners used from the skicit-learn package.

APPENDIX A - DATA

For this study we focus on attributes defined by Halstead [10] and McCabe [11]. McCabe and Halstead are module-based, where a module is the smallest unit of functionality. All our data comes from the "tera-PROMISE" repository [2]. At this time there are 14 datasets available. Since the baseline study uses 8 datasets, out of which 1 is missing, we present results for the remaining 7. We encourage researchers in the community to use metric generation tools [12][13] to generate similar data and compare their results

System	Source Language	Data Set Name
Spacecraft Instrument	C	cm1
Storage Management for Ground Data	Java	kc3
Database Application	C	mw1
Flight Software for Earth Orbiting Satellite	C	pc1
		pc2
		pc3
		pc4

Fig. 10. Data sets used in this study

using our code [3].

This data from the 7 datasets, Figure 5, was taken from 4 systems. These systems were developed in different geographical locations across North America. Within a system, the subsystem shared some common code base but did not pass personnel or code between subsystem. Each dataset was preprocessed by converting the character attribute into integer using the following rule. If class variable is "Y"/"YES" then value is 1 and if class variable is "N"/"NO" then value is 0.

The Halstead attributes were derived by Maurice Halstead in 1977. He argued that modules that are hard to read are more fault prone [10]. Halstead estimated reading complexity by counting the number of operators and operands in a module. Those were his base metrics. Based on these, he defined a list of derived metrics like length, vocabulary, difficulty, content, effort, programming time etc.

An alternative to the Halstead attributes are the complexity attributes proposed by Thomas McCabe in 1976. Unlike Halstead, McCabe argues that the complexity of pathways between modules is more insightful than just a count of symbols [11]. A module is said to have a flow-graph, i.e. a directed graph where each node corresponds to a program statement and each arc indicates the flow of control from one statement to another. Based on this flow-chart, McCabe derived metrics like cyclomatic complexity, design complexity and essential complexity.

At the end of Figure 6 are a set of miscellaneous attributes that are less well-defined than lines of code attributes or the Halstead and McCabe attributes. The meaning of these attributes is poorly documented in the baseline paper. Indeed, they seem to be values generated from some unknown tool set that was available at the time of uploading the data.

m = Mccabe		$v(g)$	cyclomatic_complexity
		$iv(G)$	design_complexity
		$ev(G)$	essential_complexity
locs	loc		loc_total (one line = one count
	loc(other)		loc_blank
			loc_code_and_comment
			loc_comments
			loc_executable
			number_of_lines (opening to closing brackets)
Halstead	h	N_1	num_operators
		N_2	num_operands
		μ_1	num_unique_operators
		μ_2	num_unique_operands
	H	N	length: $N = N_1 + N_2$
		V	volume: $V = N * log_2\mu$
		L	level: $L = V^*/V$ where $V^* = (2 + \mu_2{}^*)log_2(2 + \mu_2{}^*)$
		D	difficulty: $D = 1/L$
		I	content: $I = \hat{L} * V$ where $\hat{L} = \frac{2}{\mu_1} * \frac{\mu_2}{N_2}$
		E	effort: $E = V/\hat{L}$
		B	error_est
		T	prog_time: $T = E/18$ seconds
misc = Miscellaneous			branch_count
			call_pairs
			condition_count
			decision_count
			decision_density
			design_density
			edge_count
			global_data_complexity
			global_data_density
			maintenance_severity
			modified_condition_count
			multiple_condition_count
			node_count
			normalized_cyclomatic_complexity
			parameter_count
			pathological_complexity
			percent_comments

Fig. 11. Attributes used in the study

REFERENCES

[1] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. IEEE Transactions on Software Engineering, 33(1):213, 2007.

[2] Menzies, T., Krishna, R., Pryor, D. (2016). The Promise Repository of Empirical Software Engineering Data; http://openscience.us/repo. North Carolina State University, Department of Computer Science

[3] https://github.com/NeilBINGOHIT/fss16gNS/tree/master/project

[4] T. Menzies, J.S. DiStefeno, M. Chapman, and K. Mcgill, Metrics that Matter, Proc. 27th NASA SEL Workshop Software Eng., 2002.

[5] http://scikit-learn.org/stable/modules/generated/sklearn.naivebayes.GaussianNB.html

[6] http://i.stanford.edu/pub/cstr/reports/cs/tr/79/773/CS-TR-79-773.pdf

[7] http://machinelearningmastery.com/tactics-to-combat-imbalanced-classes-in-your-machine-learning-dataset/

[8] http://www.jair.org/papers/paper953.html

[9] http://dl.acm.org/citation.cfm?id=2578408

[10] M. Halstead, Elements of Software Science. Elsevier, 1977.

[11] T. McCabe, A Complexity Measure, IEEE Trans. Software Eng., vol. 2, no. 4, pp. 308-320, Dec. 1976.

[12] https://sourceforge.net/projects/halsteadmetricstool/?source=directory

[13] http://www.locmetrics.com/alternatives.html

[14] http://www.win.tue.nl/ aserebre/2IS55/2010-2011/10.pdf

[15] http://machinelearningmastery.com/get-your-hands-dirty-with-scikit-learn-now/