Tuning Learners before Applying: New Reflection in Defect Prediction

A, B, C

ABSTRACT

Data mining techniques have been widely applied to software defect prediction with various empirical data sets. Those proposed leaners are always evaluated against the state of the art predictors by performing statistical analysis. Undoubtedly, given the bias from data sets and accuracy indicators, the newer predictors outperform counterparts in selected experimental settings. However, most of data mining algorithm based predictors (e.g. CART, random forest, neural networks, SVM) have built-in *magic* parameters, like the number of trees in random forest algorithm. the impact of internal parameters in those methods have been neglected during evaluation. In this paper, we investigate this impact by tuning parameters in defect predictors with search-based software engineering algorithm. Specifically, we used differential evolution to tune the CART and a new predictor based on WHERE algorithm with local data, and then predictors with optimal parameters obtained from tuning process will be applied to predict defects. By comparing the performance of predictors with and without tuning process, we observe that tuning improves the predictors' performance and predictors woking with different data sets need different parameters. Our results also suggest that we should not use the predictors of the shelf with their default parameters and tuning should be a processor combined with any predictor with built-in parameters.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management—*Cost estimation*

General Terms

Experimentation, Algorithm

Keywords

Defect prediction, Data Mining, Tuning Parameters, CART, WHERE

1. INTRODUCTION

Software has becoming a large and complex system and delivering reliable and quality software is imperative for development teams. Empirical study shows that the longer the defects exist in software systems, the more the cost of time and money it will take to fix it **[need a ref]**. Therefore, project managers and software programers strive to find defects in their system as early as possible. Defect prediction has been investigated extensively in industrial and academia during the past two decades. As an important research field, building data miners [1, 2, 3, 4, 5, **?**, 6] over static code features of software system has been demonstrated to be a way to predict which models are more likely to contain defects.

Classification is an important approach to predict whether some modules in the projects are defective or non-defective. The general idea is to train the learners by using parts of data sets(e.g. ant 1.3, 1.4 in PROMISE[1]) and predict with remaining ones(ant 1.5, 1.6 and 1.7). Many types of defect predictors have been proposed based an different data mining classifiers, including CART, Random Forest [7], Naive Bayes[3],Logistic Regression [8]. During the past years, authors claimed that their new defective learner outperformed others according to their experiment and statistical analysis. To evalute those learners objectively in terms of accuracy, Lessmann et al [1] carried out a study to compare 22 classifiers over 10 public domain data sets from the NASA Metrics Data repository. By using Nemenyi's post hoc test with $\alpha = 0.05$, they concluded that the predictive accuracy of most learners didn't differ significantly in terms of the area under the receiver operating characteristics curve(AUC). Furthermore, according to the fig.2 in [1], Random Forest is significantly better than CART. Lessmann's paper motivates us to investigate whether tuning those CART's parameters by search-based software engineering method can improve the performance. Even though Lessmann considered unpruned tree and pruned tree, They didn't consider other possible parameters in CART which would have impact on the structure of trees, like the depth of the tree, the maximum and minimum number of leafs of the tree.

Software metrics are the core of all the defective prediction model. Many types of metics are used to build models, like process metrics, McCabe and Halsted metrics and CK metrics. By building prediction modes across 85 releases of 12 open source projects, Rahman et al [9] concluded that code

[1]http://openscience.us/repo/

metrics are generally less useful than process metrics for prediction. And also the code metrics don't change much from release to release and lead to stagnation in the prediction model. In [10], Radjenovi? et al [10] reviewed 106 papers regarding software prediction metrics. They found that CK objected-oriented and process metrics have been reported to be more successful in finding defects compared to traditional size and complexity metrics. Moreover, not all the CK metrics perform well equally. The best metrics from CK are CBO, WMC and RFC based on their observation. It seems that the relationship between software metrics and defective prediction is still an open question and need to be addressed. This motivates us to see : whether the impact rankings of those metrics will change after tuning parameters is applied to model learners.

What's the problem in those result?

RQ:

briefly describe our study and our result; observation

structure of this paper.

2. ALGORITHM: PREDICTOR AND TUNER

In order to conduct the experiment of this paper, we need one tool that can predict defects from the empirical data sets and a second tool to tune the built-in parameters associated with predictor. To compare the effects of the tuning process, we have three different predictors: WHERE-based Predictor, CART and Random Forest. Different Evolution(DE) as an optimizer is used as a tuner in this paper.

We choose CART and Random Forest as a predictor in this paper is motivated by [1], where the performance of both tools as predictors are significantly different based on authors' experiment as mentioned before. We'd like to investigate whether tuning can change such conclusion. WHERE-based Predictor is a new defect predictor based on WHERE[11] algorithm. A comparison with standard predictor like CART and Random Forest will better evaluate and judge the performance such new predictor. As for the tuner, there're many heuristic optimization algorithms in wild. However, DE is a good but maybe not the best candidate for the tuning process considering different performance measurements. To determine which optimizer is fitable and results in better performance is beyond this work scope. We leave it to future works. The rest of this section will describe each tool applied in this work.

2.1 WHERE-based Learner

WHERE-based Learner is composed of WHERE clustering algorithm and CART decision tree algorithm. The key idea of WHERE-based learner is that instead of training the CART decision tree based on the class labels associated with each training sample, it's using CART to build decision trees based on the cluster labels, which are generated by the WHERE clustering algorithm.

WHERE is a fast clustering algorithm designed by *menzies* for finding software artifacts with similar attributes. It clusters data on dimensions synthesized along the axis of greatest variability in the data. The way WHERE used to find such dimension is a linear-time heuristic called "FASTMAP" proposed by Faloutsos & Lin[12]. "FASTMAP" randomly picks one instance Z; find the instance *east* X that's furthest away form Z; find the instance *west* Y that's furthest away from *east* X. Next, project all the remaining points onto the line drawn between X and Y. The line $\overline{XY}$ is an approximation of the first component found by PCA.Then choose the median point as the split and recursively divide all the instances into *west* and *east* clusters until the number of instances within each cluster is less than specific minimum size. The representation of the clustering result is a tree, we call it Where-clustering tree. Such tree will be pruned if applicable in some cases. Finally, cluster labels will be assigned to instances in each of those clusters(leaves).

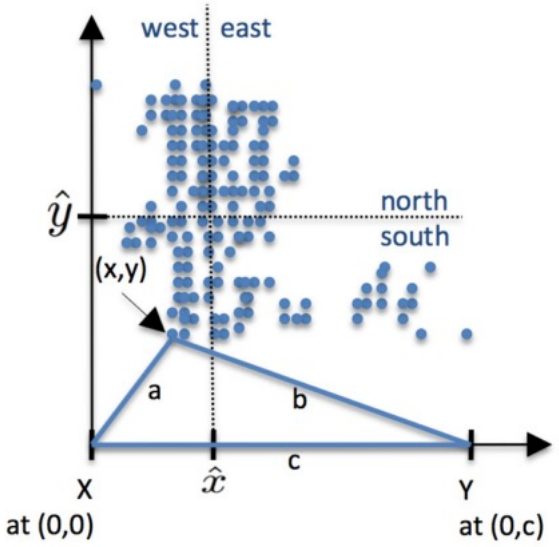

Figure 1: FASTMAP

Then based on the attributes and the cluster labels in each instances, we build a decision tree based on CART algorithm. In each leaf of the tree, there're several instances falling in. This will be the model trained for defect prediction. During testing process, a new instance comes in and traverses the tree according to values of the its attributes. If the instance could reach one leaf of the tree, the predicted value of this instance will be the mean of all the training data values associated with this leaf. Otherwise if stops at intermediate nodes of tree, the predicted value will be the mean of the all the training data value below this node. After estimating all the number of defectives in the test data, whether each instance contains defectives will be determined by comparing with a threshold value. If the predicted value is greater than threshold, the Where-based learner will predict this instance as defective otherwise non-defective.

2.2 CART

CART is an *iterative dichotomization* algorithm that finds the attribute that most divides the data such that the variance of the goal variable in each division is minimized[13]. The algorithm then recurses on each division. Finally, the cost data in the leaf divisions is averaged to generate the

estimate.

2.3 Random Forests

Breiman's website describes **Random Forests** as follows[14]. "Random Forests grows many classification trees. To classify a new object from an input vector, put the input vector down each of the trees in the forest. Each tree gives a classification, and we say the tree "votes" for that class. The forest chooses the classification having the most votes (over all the trees in the forest). Each tree is grown as follows. If the number of cases in the training set is N, sample N cases at random - but with replacement, from the original data. This sample will be the training set for growing the tree. Also, if there are M input variables, a number m<<M is specified such that at each node, m variables are selected at random out of the M and the best split on these m is used to split the node. The value of m is held constant during the forest growing. Finally, each tree is grown to the largest extent possible (there is no pruning)."

2.4 Differential Evolution Algorithm

Differential Evolution (DE)[15] is a stochastic search algorithm that optimizes a problem by iteratively trying to improve a population of candidate solutions with regard to a given quality measurement. Such method makes no assumptions about the problem being optimized and has already been used as a parameter tuner [16, 17].

DE starts with creating a population of candidate solutions and then generates new candidates based on $New = X + f * (Y - Z)$, where X, Y and Z are randomly selected solutions from the current frontier and f is a crossover factor. The newly generated solution will be added into the next generation of solutions if dominating previous old points in the frontier. To avoid meaning less iteration, early termination strategy is applied that is at the beginning, assign a value to the $life$ parameter, it would be reduced by one each time when the new generation of candidate solutions does not improve in terms of quality measurement. The DE will stop when the $life$ equals to 0.

Algorithm 1 is a list of pseudocode of DE with early termination for maximizing a score function, where np is the number of population in each generation, f is the crossover factor as mentioned, cf is the probability for crossover operation to generate new candidate, $life$ is to control termination.

Algorithm 1 Pesudocode for DE with Early Termination

Input: np, f, cf, $life$
Output: S_{best}

```
Population ← InitializePopulation(np)
S_best ←GetBestSolution(Population)
while life > 0 do
    NewGeneration ← ∅
    for i = 0 → n − 1 do
        S_i ← GenNew(Population[i], Population, cf, f)
        if Score(S_i) >Score(Population[i]) then
            NewGeneration ← S_i
        else
            NewGeneration ← Population[i]
        end if
    end for
    Population ← NewGeneration
    if ¬ Improve(Population) then
        life− = 1
    end if
    S_best ← GetBestSolution(Population)
end while
return S_best
```

3. EXPERIMENT

3.1 Data Set

The data used in this study is from PROMISE repository. Ten software defect predicition data sets are analyzed. They're *ant*, *camel*, *ivy*, *jedit*, *log4j*, *lucene*, *synapse*, *velocity*, *xalan* and *xerces*. Each of these data sets is composed of several software modules with number of defects and code attributes. For more detailed description of code attributes and the original data sets, please refer to http://openscience.us/repo/

3.2 Experiment Design

The experiment aims at investigate whether tuning helps learners improve performance in terms of accuracy measurement. We choose compare WHERE-based learner with and without tuning parameters and two *significantly different* learners Random Forest and CART according to [1]. As mentioned above, we'd like to see whether tuning will help change the rank of CART and make it comparable with Random Forest.

To evaluate accuracy performance of learners, several measurements are proposed, like probability of detection(*pd*) and probability of false alarm (*pf*)[3], the area under the receiver operating characteristics curve (AUC) [1], and precision[18]. In this work, we expect learners should identify as many defective modules as possible while avoiding false alarm. Therefore, learners are evaluated by both of *pd* and *pf* simaltanieously. A single measure, G-measure, defined as the harmonic mean of *pd* and $1 - pf$ is used. The G-measure value is between 0 and 1. The higher, the better.

$$G = \frac{2 * (1 - pf) * pd}{1 - pf + pd} \tag{1}$$

In this experiment, we use three different portions of one project data set for training, tuning and testing process. In contrast to hold out way used in [1, 3], we separate the data sets in order. Since learners are designed to predict defects in future projects, any randomly data set selection without taking into the time series will not sufficient to evaluate the performance of predicting future. To the most, that is good to evaluate the accuracy of classification but not predicting future. Since we have 10 different project data, each of which contains least 3 evolutionary versions. We use the following policy to select the data: in each project, we only use the last three data files for experiment. Specifically, the nth, $(n-1)th$, $(n-2)th$ versions of project data are selected for testing, tuning and training learners, respectively. This will make sure that we don't use the future project data to train learners and predict previous project.

To investigates the impacts of parameters on learners, we use DE as the tuner and compare the G-measure values of

Where-based learner with and without tuning, CART with and without tuning and Random Forests. Since the tuning time for Random Forests is very long, hopefully other researchers design new heuristics to speed up the tuning process for Random Forests. For the time being, even though we don't tune Random Forests, if tuning help CART outperform Random Forests or improve itself performance , we still could conclude that tuning is helpful and necessary when comparing learners.

Besides the Where-based learner implemented by ourself, we use the CART and Random Forest modules from scikit-learn [19] for this experiment. The parameters associated with different learners are listed in Fig.3. (**NEED to elaborate that there're different versions of CART, what's the point to do this experiment**). For each data set, run CART, naive Where-based learner and Random Forests with corresponding default values. Then using DE to tune corresponding parameters for CART and Where-based Learner, and run them again with the optimal parameters from tuning process to test the performance. This study ranks learners using the Scott-Knott procedure recommended by Mittas & Angelis in their 2013 IEEE TSE paper [20]. This method sorts a list of l treatments with ls measurements by their median score. It then splits l into sub-lists m, n in order to maximize the expected value of differences in the observed performances before and after divisions.

4. RESULT
5. DISCUSSION
6. RELATED WORK

Tuning in efforts estimation, software engineering.

Defect Prediction

7. THREATS TO VALIDITY

Internal and external threats

8. CONCLUSION
9. ACKNOWLEDGMENTS
10. REFERENCES

[1] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *Software Engineering, IEEE Transactions on*, 34(4):485–496, 2008.

[2] Thomas J McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308–320, 1976.

[3] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *Software Engineering, IEEE Transactions on*, 33(1):2–13, 2007.

[4] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayşe Bener. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering*, 17(4):375–407, 2010.

[5] Yue Jiang, Bojan Cukic, and Tim Menzies. Can data transformation help in the detection of fault-prone modules? In *Proceedings of the 2008 workshop on Defects in large software systems*, pages 16–20. ACM, 2008.

[6] Qinbao Song, Zihan Jia, Martin Shepperd, Shi Ying, and Jin Liu. A general software defect-proneness prediction framework. *Software Engineering, IEEE Transactions on*, 37(3):356–370, 2011.

[7] Lan Guo, Yan Ma, Bojan Cukic, and Harshinder Singh. Robust prediction of fault-proneness by random forests. In *Software Reliability Eng, 2004. ISSRE 2004. 15th Int'l Symp on*, pages 417–428. IEEE, 2004.

[8] Taghi M Khoshgoftaar and Edward B Allen. Logistic regression modeling of software quality. *International Journal of Reliability, Quality and Safety Engineering*, 6(04):303–317, 1999.

[9] Foyzur Rahman and Premkumar Devanbu. How, and why, process metrics are better. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 432–441. IEEE Press, 2013.

[10] Danijel Radjenovi?, Marjan Heri?ko, Richard Torkar, and Aleĺ t'ivkovi? Software fault prediction metrics: A systematic literature review. *Information and Software Technology*, 55(8):1397 – 1418, 2013.

[11] Tim Menzies, Andrew Butcher, David Cok, Andrian Marcus, Lucas Layman, Forrest Shull, Burak Turhan, and Thomas Zimmermann. Local versus global lessons for defect prediction and effort estimation. *Software Engineering, IEEE Transactions on*, 39(6):822–834, 2013.

[12] Christos Faloutsos and King-Ip Lin. *FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets*, volume 24. ACM, 1995.

[13] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. 1984.

[14] Leo Breiman and Adele Cutler. Random forests, 2001. https://www.stat.berkeley.edu/ breiman/RandomForests.

[15] Rainer Storn and Kenneth Price. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11(4):341–359, 1997.

[16] Mahamed GH Omran, Andries Petrus Engelbrecht, and Ayed Salman. Differential evolution methods for unsupervised image classification. In *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, volume 2, pages 966–973. IEEE, 2005.

[17] I Chiha, J Ghabi, and N Liouane. Tuning pid controller with multi-objective differential evolution. In *Communications Control and Signal Processing (ISCCSP), 2012 5th International Symposium on*, pages 1–4. IEEE, 2012.

[18] Hongyu Zhang and Xiuzhen Zhang. Comments on ?data mining static code attributes to learn defect predictors? *IEEE Transactions on Software Engineering*, 33(9):635, 2007.

[19] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[20] Nikolaos Mittas and Lefteris Angelis. Ranking and clustering software cost estimation models through a multiple comparisons algorithm. *Software Engineering, IEEE Transactions on*, 39(4):537–551, 2013.

Learner Name	Parameters	Default	Tuning Range	Description
Where-based Learner	threshold	0.5	[0.01,1]	The value to determine defective or not .
	infogain	0.33	[0.01,1]	The percentage of features to consider for the best split to build CART tree[2].
	min_sample_size	4	[1,10]	The minimum number of samples required to be a leaf for CART tree.
	min_Size	0.5	[0.01,1]	The value to determine the minimum number of samples to be a Where-clustering tree based on $n\_samples^{min\_Size}$.
	depthMin	2	[1,6]	The minimum depth of the tree below which no pruning for Where- clustering tree.
	depthMax	10	[1,20]	The maximum depth of the Where-clustering tree.
	wherePrune	False	T/F	Whether or not to prune the Where-clustering tree.
	treePrune	True	T/F	Whether or not to prune the classification tree built by CART.
CART	threshold	0.5	[0,1]	The value to determine defective or not.
	max_feature	None	[0.01,1]	The number of features to consider when looking for the best split.
	min_sample_split	2	[2,20]	The minimum number of samples required to split an internal node.
	min_smaples_leaf	1	[1,20]	The minimum number of samples required to be at a leaf node.
Random Forests	threshold	0.5	[0.01,1]	The value to determine defective or not.
	max_feature	None	[0.01,1]	The number of features to consider when looking for the best split.
	max_leaf_nodes	None	[1,50]	Grow trees with max_leaf_nodes in best-first fashion.
	min_sample_split	2	[2,20]	The minimum number of samples required to split an internal node.
	min_smaples_leaf	1	[1,20]	The minimum number of samples required to be at a leaf node.
	n_estimators	100	[50,150]	The number of trees in the forest.

Figure 2: List of parameters to be tuned in Where-based learner and CART in scikit-learn.

Learner Name	Parameters	Default	ant	camel	ivy	jedit	log4j	lucene	poi	synapse	velocity	xalan	xerces
Where-based Learner	threshold	0.5	0.16	0.87	0.89	0.68	0.58	0.62	0.06	0.09	1	0.64	0.02
	infogain	0.33	0.33	0.49	1	0.82	0.12	0.54	0.38	0.01	1	0.36	0.25
	min_sample_size	4	8	1	1	5	8	1	5	9	1	5	5
	min_Size	0.5	1	0.67	0.99	1	0.63	1	0.95	0.59	1	0.88	0.64
	depthMin	2	3	4	1	1	2	1	1	5	1	1	3
	depthMax	10	8	14	16	18	1	16	13	5	9	18	20
	wherePrune	False	True	True	False	False	False	True	True	False	True	False	True
	treePrune	True	False	True	True	True	False	False	True	False	False	False	False
CART	threshold	0.5	0.01	0.98	0.94	0.67	0.47	1	0.01	0.12	0.9	0.57	0.01
	max_feature	None	0.01	1	0.29	0.28	0.53	0.13	0.01	0.01	0.87	0.47	0.01
	min_sample_split	2	6	2	13	17	12	2	14	12	7	16	13
	min_smaples_leaf	1	6	3	18	12	3	10	4	20	8	11	1
Random Forests	threshold	0.5	0.06	0.88	1	0.73	0.41	0.81	0.11	0.16	1	0.63	0.29
	max_feature	None	0.21	0.98	0.78	0.73	0.36	0.35	0.01	0.01	0.01	0.65	0.89
	max_leaf_nodes	None	31	35	41	40	23	12	26	41	46	49	35
	min_sample_split	2	12	14	2	5	8	11	1	16	1	9	20
	min_smaples_leaf	1	6	15	2	17	3	9	19	8	2	14	9
	n_estimators	100	111	120	89	68	64	84	107	100	79	113	50

Figure 3: Optimal parameters from tuning process with objetive of G measure over different data sets.

Dataset	ant	antV1	antV2	camel	camelV1	ivy	jedit	jeditV1	jeditV2
training	20/125	40/178	32/293	13/339	216/608	63/111	90/272	75/306	79/312
tunning	40/178	32/293	92/351	216/608	145/872	16/241	75/306	79/312	48/367
testing	32/293	92/351	166/745	145/872	188/965	40/352	79/312	48/367	11/492

Figure 4: The percentage of defective instances in each experimental data set. For each experiment, training, tuning and testing data are composed of single chronological data file

Dataset	log4j	lucene	poi	poiV1	synapse	velocity	xerces	xercesV1
training	34/135	91/195	141/237	37/314	16/157	147/196	77/162	71/440
tunning	37/109	144/247	37/314	248/385	60/222	142/214	71/440	69/453
testing	189/205	203/340	248/385	281/442	86/256	78/229	69/453	437/588

Figure 5: The percentage of defective instances in each experimental data set. For each experiment, training, tuning and testing data are composed of single chronological data file

Features	ant		antV1	antV2		camel	camelV1	ivy	jedit	jeditV1	jeditV2	log4j	lucene	poi	poiV1	synapse	velocity		xerces	xercesV1	
npm																					
loc			⋆			⋆	⋆					⋆		⋆	⋆	⋆			⋆	⋆	⋆
amc														⋆	⋆				⋆	⋆	
max cc																					
lcom																					
dam	⋆		⋆	⋆		⋆		⋆	⋆	⋆	⋆		⋆	⋆	⋆	⋆	⋆		⋆		
ca																					
cbo																					
ce																					
noc																					
rfc						⋆	⋆							⋆						⋆	
dit			⋆				⋆	⋆	⋆	⋆	⋆	⋆					⋆	⋆			
mfa	⋆	⋆	⋆	⋆	⋆	⋆	⋆		⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	
cam				⋆		⋆	⋆	⋆			⋆		⋆			⋆			⋆	⋆	
avg cc																					
wmc	⋆		⋆				⋆							⋆						⋆	
moa																					
cbm									⋆	⋆	⋆	⋆			⋆		⋆				
ic	⋆			⋆				⋆	⋆	⋆	⋆		⋆				⋆	⋆			
lcom3				⋆		⋆		⋆	⋆	⋆		⋆	⋆		⋆		⋆				

Figure 6: Feature seleciton for different datasets with and without the tuning process over the objective of pd. For each data set, the stars in left and right columns are representing the features used to build defect prediction model without and with the tuning process, respectively.

Data set	median(pd) Naive_Where	Tuned_Where	Naive_CART	Tuned_CART	Naive_RanFst	Tuned_RanFst
antV0	53	100	38	100	78	97
antV1	7	100	39	100	95	100
antV2	0	100	37	100	92	100
camelV0	0	100	6	100	66	99
camelV1	80	100	46	100	80	100
ivyV0	93	100	88	100	95	100
jeditV0	89	100	66	95	96	100
jeditV1	75	100	50	100	100	100
jeditV2	45	100	36	100	100	100
log4jV0	46	100	31	100	88	100
luceneV0	81	100	48	100	98	100
poiV0	89	100	79	100	100	100
poiV1	2	100	12	100	89	100
synapseV0	0	100	28	100	90	100
velocityV0	100	100	86	100	100	100
xercesV0	64	100	46	100	78	91
xercesV1	15	100	11	100	68	87

KEY:

pd percentile ranges:

80th to 100th =
60th to 80th =
40th to 60th =
20th to 40th =

An absent bar denotes 0th to 20th percentile.

Percentiles computed separately for each data set.

Figure 7: Median pd values in tune once and test ten times experiment. Gray bars show pd values discretized into 20th percentiles ranges from min to max. All data available from http://openscience.us/repo/effort.

Features	ant	antV1	antV2	camel	camelV1	ivy	jedit	jeditV1	jeditV2	log4j	lucene	poi	poiV1	synapse	velocity	xerces	xercesV1
npm																	
loc		⋆		⋆	⋆	⋆			⋆	⋆			⋆	⋆ ⋆		⋆	⋆ ⋆
amc												⋆	⋆			⋆	⋆
max cc																	
lcom														⋆			
dam	⋆	⋆	⋆ ⋆	⋆		⋆ ⋆	⋆	⋆	⋆ ⋆		⋆	⋆	⋆ ⋆	⋆ ⋆	⋆	⋆	
ca																	
cbo						⋆											
ce																	
noc																	
rfc				⋆	⋆							⋆ ⋆		⋆		⋆	⋆
dit		⋆	⋆		⋆	⋆	⋆	⋆	⋆ ⋆	⋆			⋆		⋆		
mfa	⋆	⋆	⋆ ⋆	⋆ ⋆	⋆		⋆	⋆	⋆ ⋆	⋆	⋆ ⋆	⋆	⋆ ⋆	⋆ ⋆	⋆	⋆	⋆ ⋆
cam			⋆	⋆	⋆	⋆			⋆		⋆			⋆ ⋆		⋆	⋆ ⋆
avg cc																	
wmc	⋆	⋆			⋆							⋆		⋆			⋆
moa																	
cbm						⋆	⋆	⋆	⋆ ⋆				⋆ ⋆		⋆		
ic	⋆		⋆				⋆	⋆	⋆	⋆	⋆		⋆		⋆		
lcom3			⋆	⋆			⋆	⋆	⋆	⋆	⋆		⋆	⋆	⋆		

Figure 8: Feature seleciton for different datasets with and without the tuning process over the objective of pf. For each data set, the stars in left and right columns are representing the features used to build defect prediction model without and with the tuning process, respectively.

Data set	median(pf) Naive_Where	Tuned_Where	Naive_CART	Tuned_CART	Naive_RanFst	Tuned_RanFst
antV0	15	0	12	0	9	0
antV1	5	0	20	0	14	0
antV2	0	0	10	0	2	0
camelV0	0	0	3	0	0	0
camelV1	69	0	36	20	26	18
ivyV0	63	0	50	34	42	40
jeditV0	56	0	24	0	23	24
jeditV1	35	0	20	1	17	14
jeditV2	46	0	31	3	21	16
log4jV0	31	0	13	0	0	0
luceneV0	76	0	35	24	30	21
poiV0	70	0	38	32	34	34
poiV1	0	0	8	0	0	0
synapseV0	0	0	6	0	1	0
velocityV0	99	0	85	43	63	54
xercesV0	74	0	52	1	40	43
xercesV1	36	0	26	0	19	0

KEY:

pf percentile ranges:

80th to 100th =
60th to 80th =
40th to 60th =
20th to 40th =

An absent bar denotes 0th to 20th percentile.

Percentiles computed separately for each data set.

Figure 9: Median pf values in tune once and test ten times experiment. Gray bars show pf values discretized into 20th percentiles ranges from min to max. All data available from http://openscience.us/repo/effort.

Features	ant	antV1	antV2	camel	camelV1	ivy	jedit	jeditV1	jeditV2	log4j	lucene	poi	poiV1	synapse	velocity	xerces	xercesV1
npm											⋆			⋆			
loc		⋆		⋆	⋆			⋆		⋆ ⋆		⋆	⋆ ⋆	⋆ ⋆		⋆	⋆
amc												⋆	⋆ ⋆			⋆	⋆
max cc														⋆			
lcom													⋆				
dam	⋆	⋆	⋆	⋆		⋆ ⋆	⋆	⋆ ⋆	⋆		⋆ ⋆	⋆ ⋆	⋆ ⋆	⋆ ⋆	⋆	⋆	
ca														⋆			
cbo													⋆				
ce											⋆		⋆	⋆			
noc																	
rfc				⋆	⋆						⋆	⋆	⋆	⋆		⋆	⋆
dit		⋆			⋆	⋆ ⋆	⋆	⋆ ⋆	⋆	⋆ ⋆		⋆		⋆	⋆		
mfa	⋆	⋆	⋆	⋆	⋆		⋆	⋆ ⋆	⋆	⋆	⋆ ⋆	⋆ ⋆	⋆ ⋆	⋆ ⋆	⋆ ⋆	⋆	⋆
cam			⋆	⋆	⋆	⋆ ⋆			⋆		⋆ ⋆		⋆	⋆		⋆	⋆
avg cc											⋆		⋆	⋆			
wmc	⋆	⋆			⋆						⋆	⋆	⋆	⋆			⋆
moa														⋆			
cbm							⋆	⋆	⋆				⋆		⋆		
ic	⋆		⋆			⋆ ⋆	⋆	⋆	⋆	⋆ ⋆	⋆				⋆ ⋆		
lcom3			⋆	⋆			⋆	⋆		⋆ ⋆	⋆ ⋆		⋆ ⋆	⋆	⋆		

Figure 10: Feature seleciton for different datasets with and without the tuning process over the objective of precision. For each data set, the stars in left and right columns are representing the features used to build defect prediction model without and with the tuning process, respectively.

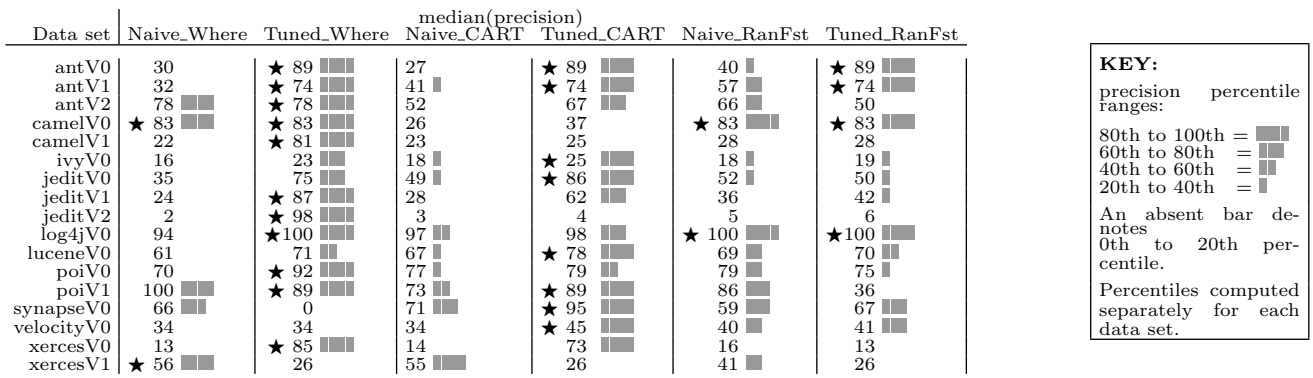

Figure 11: Median precision values in tune once and test ten times experiment. Gray bars show precision values discretized into 20th percentiles ranges from min to max. All data available from http://openscience.us/repo/effort.

Features	ant		antV1		antV2		camel		camelV1		ivy		jedit		jeditV1		jeditV2		log4j		lucene		poi		poiV1		synapse		velocity		xerces		xercesV1	
npm				⋆								⋆												⋆						⋆				
loc			⋆					⋆		⋆				⋆		⋆		⋆	⋆				⋆	⋆	⋆	⋆	⋆	⋆			⋆	⋆	⋆	⋆
amc						⋆						⋆											⋆	⋆	⋆						⋆	⋆	⋆	⋆
max cc																																		
lcom		⋆																						⋆				⋆		⋆				
dam	⋆		⋆		⋆	⋆	⋆			⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆		⋆	⋆	⋆	⋆	⋆	⋆		⋆	⋆	⋆		⋆	⋆		⋆
ca																																		
cbo		⋆										⋆												⋆										
ce		⋆												⋆																				
noc																																		
rfc		⋆						⋆		⋆	⋆			⋆		⋆				⋆				⋆				⋆				⋆	⋆	⋆
dit			⋆			⋆			⋆		⋆	⋆	⋆		⋆		⋆	⋆		⋆		⋆		⋆			⋆			⋆			⋆
mfa	⋆	⋆	⋆	⋆	⋆	⋆	⋆		⋆	⋆		⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆
cam		⋆			⋆	⋆	⋆		⋆		⋆	⋆			⋆		⋆				⋆				⋆	⋆		⋆	⋆	⋆	⋆	⋆		
avg cc														⋆																				⋆
wmc	⋆		⋆	⋆				⋆	⋆			⋆				⋆							⋆	⋆				⋆				⋆	⋆
moa																								⋆										
cbm						⋆							⋆		⋆		⋆								⋆									⋆
ic	⋆	⋆			⋆	⋆					⋆	⋆			⋆	⋆	⋆	⋆	⋆	⋆	⋆								⋆	⋆				⋆
lcom3				⋆	⋆	⋆	⋆			⋆		⋆	⋆		⋆	⋆		⋆	⋆	⋆	⋆	⋆			⋆	⋆		⋆	⋆	⋆		⋆		⋆

Figure 12: Feature seleciton for different datasets with and without the tuning process over the objective of F measure. For each data set, the stars in left and right columns are representing the features used to build defect prediction model without and with the tuning process, respectively.

Data set	median(F) Naive_Where	Tuned_Where	Naive_CART	Tuned_CART	Naive_RanFst	Tuned_RanFst
antV0	★ 39	26	32	25	25	22
antV1	11	★ 85	40	49	39	33
antV2	0	★ 86	44	51	52	56
camelV0	0	★ 28	9	★ 28	34	31
camelV1	34	★ 35	31	33	33	30
ivyV0	27	30	30	32	★ 35	★ 35
jeditV0	50	55	56	55	★ 61	60
jeditV1	37	34	36	47	45	★ 48
jeditV2	4	★ 99	5	9	9	11
log4jV0	62	7	47	★ 64	53	45
luceneV0	70	73	56	★ 75	70	★ 75
poiV0	★ 78	63	74	70	73	72
poiV1	5	★ 78	21	★ 78	76	★ 78
synapseV0	0	2	40	★ 56	52	55
velocityV0	51	51	49	51	53	★ 59
xercesV0	22	20	21	18	23	22
xercesV1	23	2	18	36	68	★ 71

KEY:

F percentile ranges:

80th to 100th =
60th to 80th =
40th to 60th =
20th to 40th =

An absent bar denotes 0th to 20th percentile.

Percentiles computed separately for each data set.

Figure 13: Median F values in tune once and test ten times experiment. Gray bars show F values discretized into 20th percentiles ranges from min to max. All data available from http://openscience.us/repo/effort.

Features	ant		antV1		antV2		camel		camelV1		ivy		jedit		jeditV1		jeditV2		log4j		lucene		poi		poiV1		synapse		velocity		xerces		xercesV1	
npm				⋆						⋆		⋆										⋆		⋆										
loc			⋆	⋆			⋆	⋆	⋆	⋆				⋆				⋆						⋆	⋆		⋆	⋆			⋆	⋆	⋆	⋆
amc								⋆		⋆		⋆											⋆	⋆	⋆			⋆	⋆	⋆	⋆		
max cc																										⋆						⋆		
lcom				⋆						⋆												⋆		⋆			⋆							
dam	⋆		⋆		⋆		⋆	⋆		⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆			⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆		
ca												⋆																						
cbo										⋆												⋆		⋆				⋆		⋆				
ce										⋆				⋆								⋆		⋆		⋆				⋆				
noc																																		
rfc		⋆		⋆			⋆	⋆	⋆	⋆		⋆		⋆		⋆				⋆		⋆	⋆					⋆		⋆	⋆		
dit			⋆	⋆				⋆	⋆	⋆	⋆	⋆	⋆		⋆	⋆	⋆		⋆				⋆				⋆	⋆		⋆			
mfa	⋆	⋆	⋆	⋆	⋆		⋆	⋆	⋆	⋆		⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆		⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆		⋆	⋆	⋆	
cam				⋆	⋆		⋆	⋆	⋆	⋆	⋆	⋆			⋆	⋆	⋆	⋆		⋆	⋆				⋆	⋆	⋆	⋆		⋆	⋆	⋆		
avg cc														⋆								⋆				⋆						⋆		
wmc	⋆		⋆			⋆		⋆	⋆	⋆		⋆				⋆		⋆		⋆		⋆	⋆	⋆				⋆		⋆	⋆		
moa				⋆																		⋆		⋆										
cbm										⋆			⋆		⋆	⋆	⋆								⋆				⋆	⋆		⋆		
ic	⋆			⋆	⋆			⋆			⋆	⋆	⋆		⋆	⋆	⋆		⋆		⋆								⋆	⋆				
lcom3				⋆	⋆		⋆	⋆		⋆	⋆	⋆	⋆		⋆	⋆			⋆	⋆	⋆	⋆			⋆	⋆			⋆	⋆		⋆		

Figure 14: Feature seleciton for different datasets with and without the tuning process over the objective of G measure. For each data set, the stars in left and right columns are representing the features used to build defect prediction model without and with the tuning process, respectively.

Data set	median(G) Naive_Where	Tuned_Where	Naive_CART	Tuned_CART	Naive_RanFst	Tuned_RanFst
antV0	65	68	53	★ 75	71	72
antV1	12	★ 60	52	★ 60	57	★ 60
antV2	0	★ 72	53	63	62	69
camelV0	0	60	10	55	60	★ 62
camelV1	45	56	53	56	★ 57	★ 57
ivyV0	53	60	63	★ 74	70	68
jeditV0	59	71	71	67	★ 75	★ 75
jeditV1	70	69	62	75	75	★ 76
jeditV2	49	62	48	61	61	★ 66
log4jV0	53	53	46	46	★ 61	57
luceneV0	37	61	55	56	61	★ 64
poiV0	45	59	66	55	★ 67	61
poiV1	5	51	22	39	★ 62	56
synapseV0	0	55	43	64	63	★ 65
velocityV0	3	56	26	★ 59	51	56
xercesV0	37	43	47	43	★ 49	48
xercesV1	26	★ 71	19	36	36	33

KEY:

G percentile ranges:

80th to 100th =
60th to 80th =
40th to 60th =
20th to 40th =

An absent bar denotes 0th to 20th percentile.

Percentiles computed separately for each data set.

Figure 15: Median G values in tune once and test ten times experiment. Gray bars show G values discretized into 20th percentiles ranges from min to max. All data available from http://openscience.us/repo/effort.