Analytics without Tuning Considered Harmful? (Two Case Studies in Defect Prediction)

Wei Fu, Tim Menzies
Computer Science, North Carolina State University, USA
wfu@ncsu.edu, tim.menzies@gmail.com

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/Subject Descriptors: D.2.8 [Software Engineering]: Product metrics; I.2.6 [Artificial Intelligence]: Induction

General Terms: Experimentation, Algorithm

Keywords: defect prediction, CART, random forests, Fastmap, bootstrap sampling, effect size, A12.

1 Introduction

> *"The common misunderstanding about science is that scientists seek and find truth. They don't- they make and test models."*
> – Neil Gershenfeld
>
> *Essentially, all models are wrong, but some are useful."*
> –George Box

How "true" are the models generated by software analytics? Suppose we use a model generated from software project data to, for example, assess the relative value of OO metrics vs procedural metrics for predicting project defects. Are we reporting "truth" in any sense of the word? And for home many other projects might we find that same "truth"?

Our reading of the
good news:

- Tuning is easy (but one measure, at least an order of magnitude easier than standard optimization problems)
- Tuning can dramatically improve the performance of a learner (in one case, from recalls of zero to 70%)

interesting news

- tunings different for different data sets.

bad news:

- Tuning also changes what was learned
  - Supposedly "bad" learners start working much better than supposedly "best" learners;
  - The models used in the tuned models were very different to those used in the untuned learners.

and we reflect on that model, are we

will it reveal the '1't A standard approach to software analytics 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

FSE '15 Bergamo, Italy

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

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 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. As shown in Figure 1, X and Y are the furthest points found by "FASTMAP" and $\overline{XY}$ is of length c. Each point in this figure has a distance a to X and distance b to Y. According to the coisne rule and Pythagoras, each instance will be mapped into 2-dimension by the following equations.

$$\begin{aligned} x &= (a^2 + c^2 - b^2)/(2c) \\ y &= \sqrt{a^2 - x^2} \end{aligned} \tag{1}$$

Then choose the median point $\hat{x}$ 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).

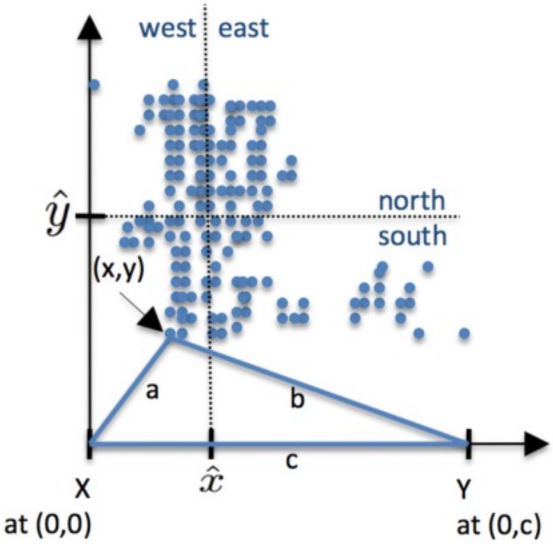

Figure 1: WHERE algoirthm

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.

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.

Algorithm 1 Pesudocode for DE with Early Termination

```
Input: $np$, $f$, $cf$, $life$
Output: $S_{best}$
$Population \leftarrow$ InitializePopulation($np$)
$S_{best} \leftarrow$GetBestSolution($Population$)
while $life > 0$ do
    $NewGeneration \leftarrow \emptyset$
    for $i = 0 \rightarrow n - 1$ do
        $S_i \leftarrow$ GenNew($Population[i], Population, cf, f$)
        if Score($S_i$) >Score($Population[i]$) then
            $NewGeneration \leftarrow S_i$
        else
            $NewGeneration \leftarrow Population[i]$
        end if
    end for
    $Population \leftarrow NewGeneration$
    if $\neg$ Improve($Population$) then
        $life- = 1$
    end if
    $S_{best} \leftarrow$ GetBestSolution($Population$)
end while
return $S_{best}$
```

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

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.2. (**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 $n\_samples^{min\_Size}$.
	wriggle	0.2	[0.01, 1]	The threshold to determine which branch in Where tree to be pruned
	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	antV0	antV1	antV2	camelV0	camelV1	ivyV0	jeditV0	jeditV1	jeditV2	log4jV0	luceneV0	poiV0	poiV1	synapseV0	velocityV0	xercesV0	xercesV1
Where based Learner	threshold	0.5	0.17	0.17	0.02	0.04	0.42	0.7	0.8	0.25	0.32	0.13	0.61	0.77	0.09	0.02	0.68	0.17	0.01
	infoPrune	0.33	0.13	0.06	0.12	0.35	0.44	0.26	0.33	0.81	0.03	0.32	0.89	0.05	0.1	0.97	0.15	0.24	0.01
	min_sample_size	4	5	2	4	1	7	6	9	6	9	8	1	3	3	9	5	4	2
	min_Size	0.5	0.49	0.24	0.41	0.26	0.12	0.03	0.26	0.06	0.19	0.38	0.02	0.07	0.2	0.2	0.32	0.13	0.86
	wriggle	0.2	0.65	0.19	0.42	0.37	0.22	0.56	0.64	0.26	0.79	0.97	0.23	0.52	0.99	0.78	0.15	0.58	0.55
	depthMin	2	4	3	2	5	5	4	3	5	4	4	2	4	2	1	4	3	1
	depthMax	10	16	7	19	11	14	16	12	12	19	17	12	13	15	13	3	17	15
	wherePrune	False	True	False	False	False	False	False	False	True	True	False	False	True	False	True	False	False	False
	treePrune	True	True	True	True	False	False	False	False	False	True	True	False	False	True	False	False	True	False
CART	threshold	0.5	0.01	0.13	0.01	0.01	0.06	0.34	0.01	0.15	0.39	0.01	0.05	0.06	0.01	0.01	0.18	0.06	0.01
	max_feature	None	0.01	0.9	0.01	0.01	0.45	0.14	0.52	0.57	0.76	1	0.68	0.01	0.33	0.01	0.73	0.04	0.01
	min_samples_split	2	17	20	7	18	9	11	15	16	12	11	2	6	3	13	19	13	10
	min_samples_leaf	1	11	1	9	2	19	14	6	15	4	17	19	9	12	5	4	7	13
Random Forests	threshold	0.5	0.09	0.01	0.01	0.01	0.36	0.49	0.08	0.01	0.01	0.01	0.05	0.01	0.01	0.01	0.2	0.15	0.01
	max_feature	None	0.01	0.63	0.01	1	0.19	0.17	0.01	0.01	0.95	0.01	0.65	0.54	0.01	0.01	0.45	0.63	0.2
	max_leaf_nodes	None	31	43	28	29	11	48	17	25	13	25	10	31	33	47	44	14	23
	min_samples_split	2	2	7	13	1	9	13	1	13	6	10	4	12	19	16	17	3	13
	min_samples_leaf	1	17	18	2	20	2	18	9	6	7	11	17	19	8	17	3	13	9
	n_estimators	100	76	64	62	141	91	67	57	104	70	67	56	59	97	81	140	85	92

Figure 3: Parameters tuned on different models over the objective of pd

Learner Name	Parameters	Default	antV0	antV1	antV2	camelV0	camelV1	ivyV0	jeditV0	jeditV1	jeditV2	log4jV0	luceneV0	poiV0	poiV1	synapseV0	velocityV0	xercesV0	xercesV1
Where based Learner	threshold	0.5	0.53	0.71	0.37	0.23	0.67	1	1	0.48	0.97	0.81	1	0.98	0.92	0.93	0.98	0.5	0.99
	infoPrune	0.33	0.68	0.97	0.43	0.01	0.22	1	0.76	0.48	0.61	0.23	0.04	0.92	0.59	0.41	0.88	0.9	0.23
	min_sample_size	4	3	1	2	1	6	4	3	5	3	9	7	3	3	8	7	2	9
	min_Size	0.5	0.07	0.03	0.36	0.06	0.21	0.38	0.15	0.19	0.29	0.08	0.35	0.24	0.38	0.67	0.01	0.25	0.58
	wriggle	0.2	0.91	0.97	0.47	0.96	0.82	0.11	0.37	0.37	0.06	0.68	0.23	0.01	0.51	0.79	0.53	0.05	0.08
	depthMin	2	4	1	3	3	1	1	3	4	2	3	3	1	1	1	1	3	1
	depthMax	10	10	15	7	4	14	3	15	16	15	18	6	6	5	18	10	6	16
	wherePrune	False	True	True	False	True	True	True	True	True	True	False	True	True	True	True	True	True	False
	treePrune	True	True	False	True	False	False	True	True	False	False	False	False	True	False	False	False	True	False
CART	threshold	0.5	1	1	0.96	0.88	1	1	0.99	0.88	1	0.83	1	1	0.92	0.9	1	1	1
	max_feature	None	0.01	1	0.93	0.58	0.01	0.73	1	0.44	0.01	0.88	0.01	0.38	0.48	0.65	0.86	0.28	0.01
	min_samples_split	2	4	4	18	3	14	19	14	16	19	12	6	2	9	8	3	6	16
	min_samples_leaf	1	12	16	19	18	1	2	5	8	8	1	10	6	12	12	13	3	10
Random Forests	threshold	0.5	0.89	0.58	0.88	0.86	1	1	1	1	1	1	1	1	0.78	0.73	1	1	0.93
	max_feature	None	0.47	0.04	0.12	0.38	0.96	0.87	0.11	0.01	1	0.16	0.85	0.99	0.78	0.77	0.01	0.35	0.23
	max_leaf_nodes	None	35	37	38	23	13	34	36	29	10	10	10	10	33	49	50	24	37
	min_samples_split	2	18	10	5	16	1	20	2	1	10	5	11	5	16	4	5	19	19
	min_samples_leaf	1	16	11	6	13	2	4	2	2	2	19	6	2	4	14	2	3	18
	n_estimators	100	103	95	76	88	68	88	136	90	144	115	150	99	110	114	53	108	99

Figure 4: Parameters tuned on different models over the objective of pf

Learner Name	Parameters	Default	antV0	antV1	antV2	camelV0	camelV1	ivyV0	jeditV0	jeditV1	jeditV2	log4jV0	luceneV0	poiV0	poiV1	synapseV0	velocityV0	xercesV0	xercesV1
Where based Learner	threshold	0.5	0.53	0.48	0.41	0.35	0.88	1	1	0.9	0.96	0.57	1	1	0.56	0.57	0.8	0.26	0.65
	infoPrune	0.33	0.68	0.74	0.31	0.45	0.78	0.31	0.53	0.85	0.04	0.73	0.54	0.15	1	0.98	0.23	0.9	0.19
	min_sample_size	4	3	6	5	2	1	5	7	1	7	5	2	8	1	1	1	2	2
	min_Size	0.5	0.07	0.23	0.2	0.21	0.25	0.54	0.18	0.36	0.28	0.51	1.0	0.33	0.5	0.84	0.83	0.04	0.21
	wriggle	0.2	0.91	0.77	0.58	0.85	0.17	0.66	0.33	0.88	0.13	0.72	0.18	0.07	0.43	0.63	0.74	0.18	0.75
	depthMin	2	4	2	5	2	4	2	2	2	4	2	3	4	1	5	1	2	1
	depthMax	10	10	14	6	13	7	16	15	14	13	19	16	9	16	14	11	13	15
	wherePrune	False	True	False	True	True	True	True	True	True	True	True	True	True	True	True	False	True	False
	treePrune	True	True	True	True	True	True	True	False	False	False	True	False	True	True	False	False	False	False
CART	threshold	0.5	0.76	0.99	0.86	0.48	1	1	1	0.71	0.62	0.65	1	0.95	0.64	0.5	1	0.99	1
	max_feature	None	0.09	0.13	0.05	0.01	0.01	0.47	0.01	0.1	0.63	0.62	0.44	0.27	0.28	0.04	0.75	0.96	0.21
	min_samples_split	2	4	19	9	9	17	14	14	16	14	12	17	3	10	12	20	13	6
	min_samples_leaf	1	15	17	8	7	1	20	1	9	12	15	10	13	7	10	4	7	20
Random Forests	threshold	0.5	0.92	0.99	0.71	0.7	1	0.82	1	1	1	0.96	0.73	1	0.76	0.33	1	1	0.98
	max_feature	None	0.23	0.13	0.46	0.69	0.37	0.56	0.71	0.01	1	0.01	0.85	0.48	0.34	0.02	0.01	0.08	0.62
	max_leaf_nodes	None	12	49	23	39	10	17	20	10	10	44	37	18	35	11	31	35	28
	min_samples_split	2	18	8	14	1	11	3	2	1	4	14	2	1	13	8	18	2	13
	min_samples_leaf	1	11	5	11	3	2	4	20	2	2	10	3	7	7	15	6	2	9
	n_estimators	100	130	146	66	96	50	50	136	84	83	129	51	150	99	58	88	85	61

Figure 5: Parameters tuned on different models over the objective of prec

Learner Name	Parameters	Default	antV0	antV1	antV2	camelV0	camelV1	ivyV0	jeditV0	jeditV1	jeditV2	log4jV0	luceneV0	poiV0	poiV1	synapseV0	velocityV0	xercesV0	xercesV1
Where based Learner	threshold	0.5	0.12	0.78	0.3	0.01	0.78	1	0.99	0.44	0.72	0.21	0.41	1	0.04	0.7	0.36	0.66	0.42
	infoPrune	0.33	0.58	0.2	0.41	0.19	0.82	0.91	0.35	1	0.85	0.46	0.24	1.0	0.85	0.36	0.48	0.42	0.82
	min_sample_size	4	6	1	1	6	8	6	7	5	1	5	8	2	7	5	3	1	1
	min_Size	0.5	0.8	0.75	0.47	0.01	1	0.64	0.99	0.43	0.23	0.47	0.72	1	0.89	0.69	0.88	0.38	0.75
	wriggle	0.2	0.21	0.7	0.83	0.25	0.55	0.01	0.63	0.93	0.43	0.33	0.52	0.32	0.72	0.1	0.43	0.34	0.1
	depthMin	2	4	5	4	3	1	6	1	4	1	1	4	1	2	2	4	5	1
	depthMax	10	16	11	5	19	8	10	14	19	5	6	6	16	3	11	5	18	12
	wherePrune	False	True	True	True	False	True	False	False	True	True	True	False	True	True	False	True	False	True
	treePrune	True	False	True	True	True	True	False	False	True	False	False	False	True	True	False	False	True	True
CART	threshold	0.5	0.01	0.62	0.13	0.01	1	0.8	0.7	0.66	0.72	0.32	0.09	0.7	0.01	0.01	0.91	0.84	0.01
	max_feature	None	0.24	0.88	0.19	0.01	0.01	0.8	0.76	0.28	0.5	0.22	0.18	0.01	0.58	0.01	0.01	0.3	0.01
	min_samples_split	2	19	3	13	5	2	10	8	9	11	12	7	10	20	5	4	14	11
	min_samples_leaf	1	15	18	11	13	3	13	9	10	15	8	15	15	7	10	10	4	1
Random Forests	threshold	0.5	0.01	0.49	0.14	0.01	1	1	1	1	1	0.66	0.53	1	0.01	0.21	1	1	0.01
	max_feature	None	0.89	0.21	0.01	0.04	0.81	0.45	0.01	0.49	0.01	0.01	0.12	0.81	0.01	0.07	0.01	0.62	0.61
	max_leaf_nodes	None	21	16	49	10	26	32	39	22	10	42	24	43	38	10	10	10	20
	min_samples_split	2	11	9	19	15	6	14	6	3	4	10	18	2	8	7	10	10	16
	min_samples_leaf	1	6	11	13	7	16	4	2	18	2	6	19	2	9	4	17	2	5
	n_estimators	100	88	99	124	148	56	101	116	55	122	112	75	55	92	129	58	107	121

Figure 6: Parameters tuned on different models over the objective of F

Learner Name	Parameters	Default	antV0	antV1	antV2	camelV0	camelV1	ivyV0	jeditV0	jeditV1	jeditV2	log4jV0	luceneV0	poiV0	poiV1	synapseV0	velocityV0	xercesV0	xercesV1
Where based Learner	threshold	0.5	0.16	0.44	0.07	0.01	0.5	0.9	0.75	0.5	0.7	0.38	0.68	1	0.01	0.15	1.0	0.84	0.01
	infoPrune	0.33	0.54	1.0	0.07	0.55	0.79	0.74	0.36	0.47	0.65	0.27	0.69	0.9	0.73	0.44	0.73	0.87	0.01
	min_sample_size	4	8	9	1	9	1	5	7	1	8	5	2	1	3	4	3	1	6
	min_Size	0.5	0.76	0.71	0.91	0.42	0.63	0.64	0.88	0.45	1	0.92	0.94	0.9	0.4	0.55	0.68	0.49	0.65
	wriggle	0.2	0.57	0.11	0.79	1	0.5	0.25	0.51	0.28	0.46	0.06	0.61	0.28	0.01	0.61	0.57	0.65	0.02
	depthMin	2	4	1	1	6	4	2	2	3	3	1	1	6	1	4	2	5	2
	depthMax	10	8	6	4	9	9	18	4	10	6	6	7	20	9	16	14	19	6
	wherePrune	False	True	True	True	False	True	True	True	True	True	True	True	True	True	False	False	True	True
	treePrune	True	True	False	True	False	False	False	False	False	True	False	False	False	True	True	True	False	True
CART	threshold	0.5	0.17	0.47	0.01	0.01	0.88	0.59	0.98	0.49	0.45	0.99	0.69	1	0.01	0.01	0.95	0.69	0.01
	max_feature	None	0.01	0.28	0.61	0.01	0.87	0.98	0.77	0.83	0.01	0.6	0.01	0.78	0.29	0.01	0.81	0.72	1
	min_samples_split	2	18	17	12	2	2	18	3	10	15	12	2	10	18	9	8	10	13
	min_samples_leaf	1	11	18	14	15	1	10	2	18	12	20	7	1	1	11	9	1	1
Random Forests	threshold	0.5	0.35	0.42	0.17	0.01	0.78	1	0.86	0.39	0.71	0.42	0.87	1	0.06	0.06	1	1	0.2
	max_feature	None	0.13	0.98	0.01	0.36	0.53	0.79	0.98	1	0.01	0.12	0.75	0.82	0.79	0.64	0.01	0.71	0.8
	max_leaf_nodes	None	13	22	42	18	10	43	23	37	49	47	15	10	45	16	36	13	39
	min_samples_split	2	9	8	13	1	10	14	10	1	1	4	6	1	5	14	1	8	12
	min_samples_leaf	1	3	14	15	2	17	6	5	2	14	2	9	3	18	16	2	2	4
	n_estimators	100	107	131	114	125	138	138	57	84	50	58	83	70	83	121	150	150	132

Figure 7: Parameters tuned on different models over the objective of G

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 8: 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 9: 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 10: 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.

	median(pd)					
Data set	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 11: 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 12: 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.

			median(pf)			
Data set	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 13: 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 14: 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.

			median(precision)			
Data set	Naive_Where	Tuned_Where	Naive_CART	Tuned_CART	Naive_RanFst	Tuned_RanFst
antV0	30	★ 89 ▮▮▮▮	27	★ 89 ▮▮▮▮	40 ▮	★ 89 ▮▮▮▮
antV1	32	★ 74 ▮▮▮▮	41 ▮	★ 74 ▮▮▮▮	57 ▮▮	★ 74 ▮▮▮▮
antV2	78 ▮▮▮▮	★ 78 ▮▮▮▮	52	67 ▮▮▮	66 ▮▮	50
camelV0	★ 83 ▮▮▮▮	★ 83 ▮▮▮▮	26	37	★ 83 ▮▮▮▮	★ 83 ▮▮▮▮
camelV1	22	★ 81 ▮▮▮▮	23	25	28	28
ivyV0	16	23 ▮▮▮	18 ▮	★ 25 ▮▮▮▮	18 ▮	19 ▮
jeditV0	35	75 ▮▮▮	49 ▮	★ 86 ▮▮▮▮	52 ▮	50 ▮
jeditV1	24	★ 87 ▮▮▮▮	28	62 ▮▮▮	36	42 ▮
jeditV2	2	★ 98 ▮▮▮▮	3	4	5	6
log4jV0	94	★100 ▮▮▮▮	97 ▮▮	98 ▮▮▮	★ 100 ▮▮▮▮	★100 ▮▮▮▮
luceneV0	61	71 ▮▮	67 ▮	★ 78 ▮▮▮▮	69 ▮▮	70 ▮▮
poiV0	70	★ 92 ▮▮▮▮	77 ▮	79 ▮▮	79 ▮▮	75 ▮
poiV1	100 ▮▮▮▮	★ 89 ▮▮▮▮	73 ▮▮	★ 89 ▮▮▮▮	86 ▮▮▮	36
synapseV0	66 ▮▮▮	0	71 ▮▮▮	★ 95 ▮▮▮▮	59 ▮▮▮	67 ▮▮▮
velocityV0	34	34	34	★ 45 ▮▮▮▮	40 ▮▮	41 ▮▮▮
xercesV0	13	★ 85 ▮▮▮▮	14	73 ▮▮▮▮	16	13
xercesV1	★ 56 ▮▮▮▮	26	55 ▮▮▮▮	26	41 ▮▮	26

KEY:

precision 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 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 16: 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.

	median(F)					
Data set	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 17: 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 18: 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.

	median(G)					
Data set	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 ▮

Figure 19: 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.

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.

Datasets	Tuned_Where	Naive_Where	Tuned_CART	Naive_CART	Tuned_RanFst	Naive_RanFst
ant	241.5	12.6	7.4	0.8	13.3	1.4
antV1	347.5	23.5	11.6	0.8	19.5	2.0
antV2	696.5	60.0	23.6	1.5	22.6	3.1
camel	1412.6	76.5	34.7	1.7	51.1	3.0
camelV1	3391.9	230.6	36.1	2.4	46.3	6.9
ivy	93.9	9.9	5.1	0.6	9.7	1.6
jedit	679.2	51.1	12.7	0.8	18.1	2.8
jeditV1	1078.5	64.9	17.2	0.9	24.6	3.0
jeditV2	838.0	68.2	16.4	1.1	23.4	3.4
log4j	190.8	13.4	5.2	0.4	11.0	1.4
lucene	421.0	27.6	9.2	0.7	13.3	2.3
poi	348.6	40.4	10.6	0.9	21.7	2.7
poiV1	1060.7	69.4	20.8	1.0	26.3	2.6
synapse	350.9	18.5	9.5	0.5	14.4	1.5
velocity	257.0	28.3	7.5	0.5	13.3	1.9
xerces	239.9	21.4	14.8	0.8	15.2	2.1
xercesV1	2433.9	130.3	27.6	1.4	25.3	3.7

Figure 20: Time (in seconds) spent on different models over the objective of pd

Datasets	Tuned_Where	Naive_Where	Tuned_CART	Naive_CART	Tuned_RanFst	Naive_RanFst
ant	33.3	14.1	2.4	0.7	2.6	1.7
antV1	59.3	25.5	4.2	0.9	3.0	2.2
antV2	131.4	67.4	3.0	1.7	4.2	3.7
camel	167.8	85.7	3.9	2.0	5.1	3.5
camelV1	503.5	325.1	68.5	3.6	146.6	8.8
ivy	116.4	11.5	8.5	1.3	23.9	1.9
jedit	336.6	59.9	6.7	0.9	29.3	3.4
jeditV1	159.9	79.2	2.1	1.2	36.9	3.5
jeditV2	151.4	80.0	21.8	1.4	38.6	3.9
log4j	37.4	15.5	1.9	0.5	5.0	1.8
lucene	436.4	33.9	17.0	1.0	49.4	3.2
poi	202.7	48.5	24.6	1.3	50.4	3.5
poiV1	184.3	79.9	2.5	1.5	5.5	3.2
synapse	36.9	20.5	1.4	0.7	2.7	1.7
velocity	157.9	31.6	10.4	0.7	33.2	2.2
xerces	59.5	23.5	19.6	1.2	31.0	2.4
xercesV1	286.6	151.8	8.0	1.8	5.4	4.5

Figure 21: Time (in seconds) spent on different models over the objective of pf

Datasets	Tuned_Where	Naive_Where	Tuned_CART	Naive_CART	Tuned_RanFst	Naive_RanFst
ant	158.6	13.3	8.8	0.7	18.1	1.6
antV1	298.9	25.4	13.5	0.9	15.8	2.2
antV2	632.0	65.0	14.4	1.7	29.3	3.6
camel	566.3	94.3	26.0	1.8	20.8	5.5
camelV1	4954.0	243.0	52.1	2.3	42.7	7.2
ivy	148.7	10.0	8.6	0.6	11.0	1.6
jedit	972.5	51.9	15.8	0.8	22.3	2.9
jeditV1	1102.5	73.6	19.7	0.9	46.5	3.5
jeditV2	837.5	76.9	15.7	1.3	65.3	3.7
log4j	289.2	17.1	6.8	0.6	17.1	1.6
lucene	471.6	29.6	8.7	0.8	16.8	2.5
poi	651.4	52.0	11.4	1.1	23.1	3.3
poiV1	1666.8	86.2	13.8	1.4	25.7	3.2
synapse	209.2	23.7	9.2	0.7	15.8	1.7
velocity	345.5	33.5	11.5	0.7	12.7	2.3
xerces	392.2	27.2	13.9	1.2	43.0	3.0
xercesV1	1619.8	157.1	23.8	1.7	34.8	5.0

Figure 22: Time (in seconds) spent on different models over the objective of prec

Datasets	Tuned_Where	Naive_Where	Tuned_CART	Naive_CART	Tuned_RanFst	Naive_RanFst
ant	302.5	13.3	6.9	0.7	13.8	1.6
antV1	328.7	27.2	12.7	0.8	15.4	2.0
antV2	1030.2	73.0	31.3	1.7	34.6	4.2
camel	1838.4	92.5	33.4	2.0	43.7	3.4
camelV1	4352.4	275.8	46.2	5.4	42.8	7.8
ivy	167.9	11.9	4.1	0.7	18.8	1.8
jedit	830.6	62.9	7.8	1.0	45.7	3.3
jeditV1	1055.2	81.8	13.6	1.1	30.7	3.6
jeditV2	1322.7	92.2	18.1	1.3	35.3	4.1
log4j	370.7	16.3	5.6	0.5	16.1	1.7
lucene	385.3	31.9	8.8	0.8	21.3	2.6
poi	720.7	46.2	10.2	0.9	19.9	3.0
poiV1	1229.0	79.4	14.9	1.2	25.9	2.9
synapse	377.0	21.6	10.3	0.6	21.0	1.7
velocity	275.6	39.6	8.9	0.7	13.2	2.1
xerces	310.7	25.4	13.3	0.9	21.6	2.3
xercesV1	1792.1	147.9	37.1	1.6	35.4	4.0

Figure 23: Time (in seconds) spent on different models over the objective of F

Datasets	Tuned_Where	Naive_Where	Tuned_CART	Naive_CART	Tuned_RanFst	Naive_RanFst
ant	164.0	13.9	9.2	0.9	14.5	1.5
antV1	560.3	27.3	15.5	1.0	17.2	2.4
antV2	1743.9	77.3	18.9	1.7	25.5	3.6
camel	1823.1	93.0	39.8	1.9	39.1	3.4
camelV1	6679.7	234.3	39.7	2.2	74.5	7.0
ivy	261.9	10.9	9.9	0.6	17.4	1.6
jedit	993.8	54.4	11.1	0.8	23.7	2.8
jeditV1	1110.0	68.5	12.3	0.9	27.2	3.0
jeditV2	858.2	72.5	15.4	1.1	23.3	3.4
log4j	247.7	15.2	3.8	0.4	9.1	1.4
lucene	610.9	30.6	6.1	0.7	28.7	2.4
poi	1466.6	47.4	18.3	0.9	33.0	2.7
poiV1	1365.6	77.4	17.2	1.1	31.1	2.7
synapse	202.6	21.9	8.0	0.5	19.8	1.5
velocity	869.8	31.4	10.5	0.6	29.8	1.9
xerces	534.1	27.1	12.6	0.8	24.1	2.0
xercesV1	2825.7	143.5	22.4	1.5	48.1	3.8

Figure 24: Time (in seconds) spent on different models over the objective of G

Features	Pd		Pf		Precision		F		G		SUM
noc	0	0	0	0	0	0	0	0	0	0	0
ca	0	0	0	0	0	1	0	0	0	1	2
max_cc	0	0	0	0	0	1	0	0	0	2	3
moa	0	0	0	0	0	1	0	1	0	3	5
avg_cc	0	0	0	0	0	3	0	2	0	4	9
cbo	0	0	0	1	0	1	0	3	0	5	10
npm	0	0	0	0	0	2	0	4	0	5	11
lcom	0	0	0	1	0	1	0	4	0	5	11
ce	0	0	0	0	0	3	0	2	0	6	11
amc	4	0	4	0	4	1	4	5	4	6	32
cbm	6	0	6	2	4	1	4	2	5	4	34
rfc	4	0	4	3	4	4	4	9	4	11	47
wmc	5	0	5	1	5	3	5	7	5	12	48
ic	8	1	8	1	9	3	8	8	9	5	60
dit	8	1	8	3	8	5	7	8	8	8	64
cam	9	0	9	2	9	3	9	8	9	11	69
loc	9	1	8	4	9	4	9	8	8	10	70
lcom3	9	0	8	2	8	5	8	13	9	10	72
dam	14	0	14	5	14	6	14	12	14	12	105
mfa	16	3	16	7	16	6	16	16	16	13	125

Figure 25: Counts of features selected by different goals. For each goal, the numbers in right and left columns represent the counts of features selected for all the data sets.