Impact of Parameter Tuning and Ensemble Methods on Fault Prediction Models

Guru Darshan Pollepalli Manohara
North Carolina State University,
Raleigh, NC
USA
gpollep@ncsu.edu

Abstract

Fault prediction models provide indispensable information to organizations and help them to redirect their efforts and resources to solving the issues in fault prone modules. Menzies et al [1] have shown that tuning a learner can provide great performance improvements. This paper augments the algorithm with ensemble learning which will be tuned for a specific dataset and goal to examine if the performance can be improved by tuning.

CCS Concepts •**Machine Learning** → Ensemble methods; Parameter tuning; Classification •**Software Engineering** → Fault prediction

Keywords Fault Prediction, Ensemble Methods, Parameter Tuning

1 INTRODUCTION

Businesses all over the world, heavily invest in Software Quality Testing to deliver quality end products to the customers. Many businesses believe that the cost required to rectify is higher when the product reaches the customer than when it is in testing or development phase. A plethora of software products are punctuated with defects today. The quality of the product is an essential driving characteristic and companies which adhere to the quality standards boost customer confidence, which in turn translates into higher returns for their investors and shareholders.

The application of machine learning algorithms can greatly accelerate defect prediction in software modules. With the advent of computation intensive hardware, the task of defect prediction can be accelerated without compromising on the accuracy of the model. Yet most of the models are reliable only up to a certain extent. A plethora of factors causes these limitations. The factors can range from simple noise in the dataset to highly inaccurate models. Menzies et al [1] demonstrated the magic of parameter tuning on learners and exposed the dangers of using untuned or off-shelf tuned models for defect prediction.

The main aim of this paper is to compare and contrast the performance of untuned learners with tuned and ensemble learners. Given that there are many datasets considered in this paper for evaluation, the models and results generated by each experiment are purely a function of the dataset under consideration and the goal for which the learner was optimized. That is, the results vary from model to model and from goal to goal. Details about datasets, learners, parameters (used for tuning) and the ensemble techniques are discussed in further sections. To summarize, the experiment can be segregated into four different test scenarios as shown below:

TS1 – Evaluate the Learners' performance using only their default parameter settings and without using Ensemble methods or Tuning algorithms

TS2 – Evaluate Learners' performance with the application of Tuning algorithms.

TS3 – Evaluate Learners' performance using ensemble methods.

TS4 – Evaluate the performance of Learners with the application of Tuning algorithms for parameter optimization and Ensemble Methods.

For **TS4,** an (algorithm) ensemble will consist of multiple learners. Each learner will contain its own set of parameters. If there are 5 learners (which are part of an ensemble) each with 4 parameters. Then the combined ensemble, containing all the learners must be tuned for 20 parameters. This might increase the time required for convergence of the tuning algorithm. Also, the parameter space to be explored by the tuning algorithm is consequently scaled up by a factor proportional to the number of learners in the ensemble. To examine the performance benefits and drawbacks of this methodology, **TS4** was further divided into two sub-parts:

TS4a – Evaluate the performance of the ensemble after tuning with Differential Evolution (Fig 1)

TS4b – Evaluate the performance of an ensemble learner composed of tuned learners from **TS2** (Fig 2).

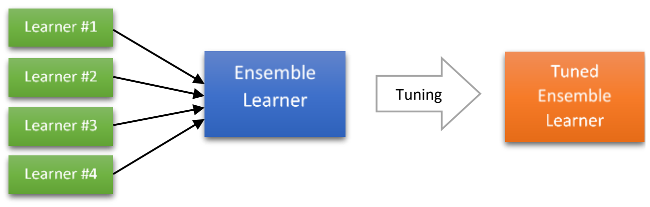


Figure 1. TS4a Visualization

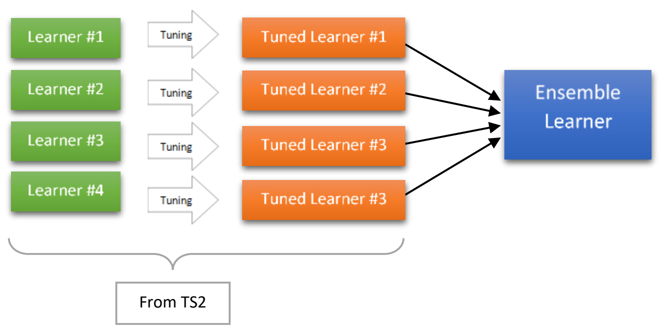


Figure 2. TS4b Visualization

As suggested before, the results obtained in each of the test scenarios (**TSx**) are highly dependent on the Dataset under consideration and the Goal for which the learner was optimized. As suggested by Menzies et al [1], the application of off-shelf tuned learners can have detrimental impact on the performance. In case of TS4b, though the learners obtained from TS2 can be considered as off-shelf tuned learners, the tuning is done specifically for a given Dataset and Goal. Hence, the context of the tuning plays a significant role while evaluating the performance of the learners.

With the Test Scenarios as described above, this paper attempts to answer the following Research Questions:

RQ1: Does the application of ensemble methods (**TS3**) on the data learning algorithms always result in better performance than defaults (**TS1**)?

Result - No, untuned ensembles did not provide the best performance when compared to individual learners. The performance was only slightly better than the poorly performing learner in the list.

RQ2: Does tuning the parameters for learners (**TS2**) always result in better performance compared to defaults (**TS1**)?

Result - Yes, tuning improved the performance in most of the cases (60-70%). Tuning was greatly beneficial for all the datasets in the experiment.

RQ3: Does the combination of ensemble methods and parameter tuning (**TS4**) always yield better results compared to direct application of ensemble methods or parameter tuning?

Result – Tuning ensemble resulted in better results for 3 datasets when tuned for accuracy. No improvement over the best individual learner when tuned for F1-score.

For **RQ3,** the performance can be evaluated based on the value of the goal generated by the ensemble method and the time taken by the procedure. Specifically, the time consumed by **TS4a** versus **TS4b** (for a given dataset and goal) can be compared. In summary, this paper tries to provide the values for Table 1.

Table 1. Performance of Tuned vs Untuned learners

	Untuned	Tuned
Single Learners*	< From TS1 >	< From TS2 >
Ensemble Learner	< From TS3 >	< From TS4a and TS4b >

Paper Organization

The remainder of the paper is organized as follows. Section 2 describes the motivation and a brief overview of defect prediction in the field of Software Engineering. Section 3 illustrates experimental design and the evaluation methodology. Section 4 describes the results of the experiment. Section 5 provides Threats to Validity for the paper. Section 6 and 7 provide Conclusion and Future Work respectively. Section 8 describes the flow of execution in the code and how to run it.

2 BACKGROUND AND MOTIVATION

There has been a recent sprout in interest for application of machine learning algorithms for finding defects in software products. These models can use the previous knowledge of the products to provide a testing framework for defect detection with a very high accuracy. Identification of fault prone modules at early stages can prevent huge loses to the company. In the field of Software Engineering, majority of the literature is directed at fault prediction and project planning. Menzies et al [12] suggest that the probability associated with the identification of a bug using fault prediction models is relatively high (71%) when compared to traditional code review methodologies which result in a probability of around 60%. Human errors can easily creep in when the code review is performed manually for defect detection. Application of fault prediction models can significantly improve the odds of finding issues with higher accuracies.

This project heavily relies on the Object-Oriented metrics postulated by Chidamber and Kemerer [2]. Computing CK metrics would involve correlating the defects from a bug repository like Bugzilla or GitHub and map them to the corresponding classes responsible for the bugs. Tibor et al [10] explored Bugzilla repository to obtain bug reports of projects associated with Mozilla and manually examined nearly 3000 bug reports to extract the classes responsible for the faults. All the datasets used in this project were downloaded from PROMISE data repository which provide CK metrics for a plethora of open-source projects. The widespread use of CK metrics and other Object-oriented metrics was investigated by Danijel et al [6]. Their paper concluded that nearly 49% of the metrics used were Object-oriented metrics. Alternatively, the application of class level metrics was explored by Koru et al [7] where they showed that class level data yielded better performance.

Sub-optimal setting for control parameters can reduce the performance of learners. Chakkrit et al [8] have provided evidence showing the negative impacts of having an untuned fault prediction model. The generated model is highly reliant on the control parameters used at the time of model training. The value of K in KNN or the number of estimators in a Random Forest can have significant impact on the performance of the model. Low values of K in KNN would cause overfitting of data whereas higher value of K would cause underfitting of data. Models which are overfit are very prone to noise and hence disadvantageous to the fault detection ability of the model. Jiang et al [9] show that the default control parameter values of Random Forests and Naïve Bayes are often suboptimal.

Menzeis et al [1] suggest that tuning is an under-explored optimization problem in the field of data learning algorithms. This project aims to extend the ideas presented in the paper to other learning algorithms and combine them with ensemble methods to perform a comparative analysis on the results obtained.

3 EXPERIMENTAL DESIGN

This section mainly deals with setup and evaluation of the experiment. The first subsection delineates the setup and provides a brief overview of the individual components that are part of the experiment (like datasets, learners, performance parameters). The second subsection provides an in-depth overview of all the steps used in the evaluation criteria and methodologies implemented to obtain the results.

3.1 Setup

All the datasets considered for experimentation contain metrics defined by Chidamber and Kermer [2] for object oriented design paradigm. The list metrics considered are Weighted Methods per Class (**WMC**), Depth of Inheritance Tree(**DIT**), Number of Children (**NOC**), Coupling Between Object Classes (**CBO**), Response for a Class (**RFC**), Lack of Cohesion of Methods (**LCOM**), etc. These datasets are associated with different open-source projects and were downloaded from PROMISE data repository. All the chosen datasets have data for atleast 3 versions. This requirement is in parallel with the requirement of Menzies et al [1] where the first version of the dataset is used within the scoring function to train the model, second version is used as test dataset to obtain the value of the “goal” and the third version will be used to evaluate the performance of the model after tuning. Fig 3 illustrates the role of individual versions in the experiment.

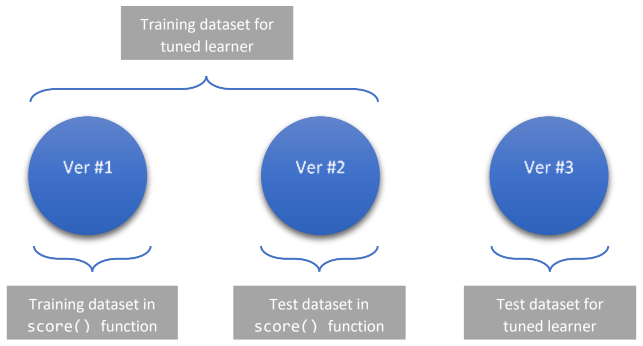

Training dataset for tuned learner
Ver #1
Ver #2
Ver #3
Training dataset in score() function
Test dataset in score() function
Test dataset for tuned learner

Figure 3. Role of three versions in tuning

With the above requirement, the datasets listed in Table 2 were considered for examination.

Table 2. Datasets and versions for experimentation

Dataset	Version #1	Version #2	Version #3
ivy	1.1	1.4	2.0
jedit1	3.2	4.0	4.1
jedit2	4.0	4.1	4.2
jedit3	4.1	4.2	4.3
lucene	2.0	2.2	2.4
velocity	1.4	1.5	1.6
xalan1	2.4	2.5	2.6
xalan2	2.5	2.6	2.7
xerces	1.2	1.3	1.4

Table 3 lists all the learners used in the experiment. The list of parameters associated with each learner are also listed. In Table 3, it can be seen that Naïve Bayes is tuned on only one parameter. Naïve Bayes was only included to improve the performance of the ensemble. Hence all the tuned and untuned results associated with Naïve Bayes will be identical.

Table 3. List of learners and their parameters

Learner	Parameters
CART	max_features
	max_depth
	min_samples_split
	min_samples_leaf
Random Forest	max_features
	max_leaf_nodes
	min_samples_split
	min_samples_leaf
	n_estimators
Naïve Bayes	priors
SVM	kernel
	C
K Nearest Neighbors	n_neighbors
	weights

3.2 Performance Criteria

The ensemble method considered in this experiment is a Voting Classifier which uses weight based voting technique. The performance parameters considered in this paper are **F1-score** and **Accuracy.** The confusion matrix is shown in Table 4.

Table 4. Confusion Matrix

	Predicted Positive	**Predicted Negative**
Actual Positive	TP	FN
Actual Negative	FP	TN

Accuracy is a measure of the number of samples from the test dataset which were correctly predicted by the learner. Accuracy can be computed using the formula:

$$Accuracy = \frac{TP + TN}{TP + FN + TN + FP}$$

Recall measures the number of relevant items classified correctly by the learner. **Precision** is the measure of the number of relevant results [11]. **F1-score** is defined as the harmonic mean of Precision and Recall.

$$Recall = \frac{TP}{TP + FN}$$

$$Precision = \frac{TP}{TP + FP}$$

$$F1\ score = \frac{2 * Precision * Recall}{Precision + Recall}$$

3.3 Evaluation Methodology

This sub-section extends on the experimental setup defined in the previous sub-section. The experimental methodology follows a chained approach to test and evaluate the results of the experiment. To simplify the complexity in combining the results associated with individual test scenarios **TSx** ($\boldsymbol{x = \{1, 2, 3, 4a, 4b\}}$), the experiment was executed in a five-step process to analytically compare the performance of the learners after each step. The steps are outlined in Figure 4.

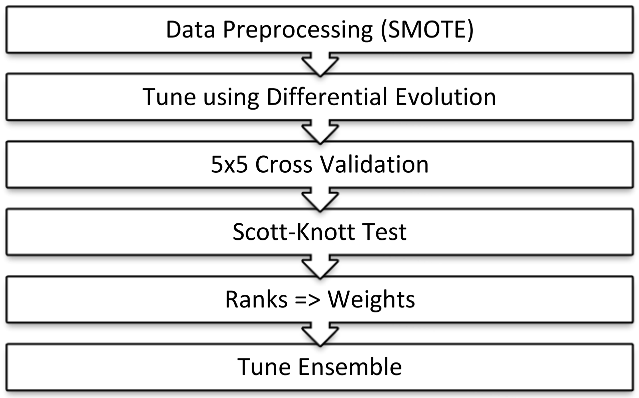


Figure 4. Steps in Evaluation Methodology

3.3.1 Data Preprocessing

All the datasets used in the experiment list the number of bugs associated per 'Java class'. To reduce the number of labels in the classification task at hand, the data was preprocessed to label all the rows of data with zero bugs as **0** and all the remaining rows as **1**. In summary, only two class labels existed: **i) Bug free (zero bugs)** and **ii) Bug prone (bugs > 0).**

Most of the datasets are riddled with the issue of imbalance in classification label distribution where the number of samples belonging to one class are significantly greater than the number of samples from other classes. As evidenced by Rahul et al [3], the data imbalance is severe in datasets like **jedit** and **ivy.** A similar result was obtained when the data imbalance was evaluated for the training data (Version#1 and Version#2 as shown in Fig 3) and the results are shown in Fig 5. It was observed that most of the datasets like **ivy, jedit1, jedit2, jedit3** and **xerces** have data imbalance issues. To mitigate the negative effects posed by data imbalance, SMOTE was applied on the datasets for better performance. SMOTE was only applied on the training dataset and not test data. As explained by Nadeem et al [4], SMOTE might not always provide the best possible results. They propose that under-sampling (of majority class) techniques lead to better results.

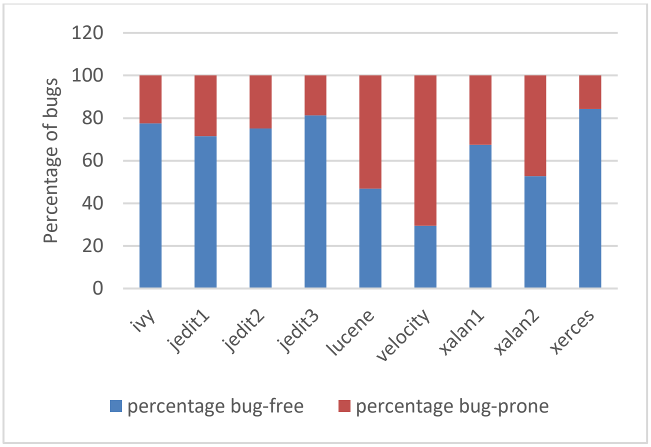


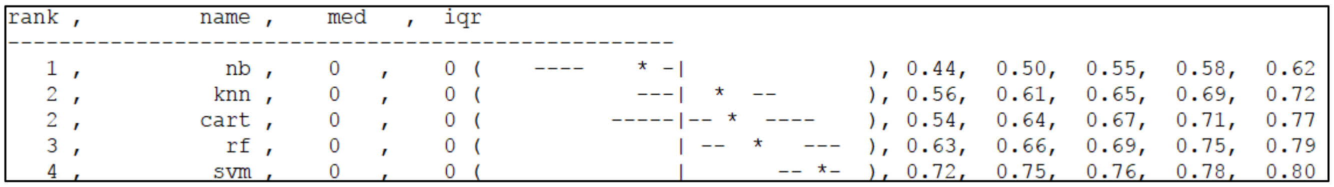
Figure 5. Data imbalance across datasets

3.3.1 Tuning using Differential Evolution

After applying SMOTE on the datasets to remove data imbalance between the classes, the datasets were used to tune the learners (Table 3) individually using the methodology outlined in Fig 3. For a given candidate (containing values for parameters) from the frontier, the first version of the dataset will be used to train the model in the `score()` function of Differential Evolution algorithm (as outlined in Menzies et al [1]). Number of generations were large enough to accommodate convergence of the value of *goal* at hand. If the value of the *goal* was observed to fluctuate during the final generations, then the *number of generations* were manually increased to allow for the convergence of goal values. To enable faster convergence, the probability of crossover Cr was set to 0.8. The result for tuning Lucene dataset for F1-score is shown in Fig 6 and the corresponding parameters for the learners which resulted in the best value of F1-score are listed in Table 5.

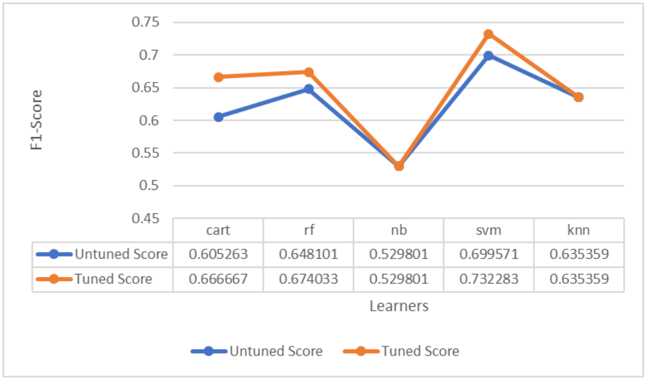
Table 5. Parameter value of learners for best F1-score. (Lucene)

Learner	Best Parameters
cart	`max_features= 0.42, min_samples_leaf= 20, min_samples_split= 7, max_depth= 3`
rf	`max_leaf_nodes= 50, min_samples_leaf= 1, max_features= 0.43, n_estimators= 57, min_samples_split= 13`
nb	`priors= None`
svm	`kernel= rbf, C= 10.0`
knn	`n_neighbors= 4, weights= distance`



```
rank ,          name ,    med   ,  iqr
----------------------------------------------------
   1 ,            nb ,    0    ,    0 (    ----     * -|                ), 0.44,  0.50,  0.55,  0.58,  0.62
   2 ,           knn ,    0    ,    0 (             ---|  *  --         ), 0.56,  0.61,  0.65,  0.69,  0.72
   2 ,          cart ,    0    ,    0 (           -----|-- *  ----      ), 0.54,  0.64,  0.67,  0.71,  0.77
   3 ,            rf ,    0    ,    0 (                | --   *    ---  ), 0.63,  0.66,  0.69,  0.75,  0.79
   4 ,           svm ,    0    ,    0 (                |         -- *-  ), 0.72,  0.75,  0.76,  0.78,  0.80
```

Figure 6. Scott-Knott test output (stats.py) for Lucene (F1-Score)



	cart	rf	nb	svm	knn
Untuned Score	0.605263	0.648101	0.529801	0.699571	0.635359
Tuned Score	0.666667	0.674033	0.529801	0.732283	0.635359

Figure 7. F1-scores of Tuned vs Untuned Learners for Lucene dataset

3.3.2 Weights Assignment for Learners

Fig 1 illustrates the process for creating the ensemble learner. Initially, it was decided that all the learners which were part of the ensemble would have equal weight while voting. But during experimentation, it was observed that learners like Naïve Bayes (in the case of F1-score of Lucene as depicted in Fig 6) performed poorly within the ensemble and assigning equal weight to it when compared to better performing learners like SVM and Random Forests would create a poor ensemble learner whose best performance (after tuning) was only slightly better than the lowest two learners in the ensemble. To alleviate this issue, different weights had to be assigned to the individual learners in the ensemble.

Upon arranging the learners in the decreasing order of their F1-scores (from Fig 6), it can be observed that

$$SVM > Random\ Forest > CART > KNN > Naive\ Bayes$$

But we can't use this order directly and assign weights based on it. Assigning SVM with a weight 5, Random Forest with a weight of 4 and so on would not be fair. Upon closer examination of Fig 6, it can be observed that the F1-scores of CART, Random Forest and KNN are very close to each other. There was need for a way to statistically tell them apart and rank all the five learners based on their performance. Eventually these ranks could be used to compute weights (proportional to their ranks) and create a weight based ensemble learner. **Scott-Knott test** perfectly fit the requirement. To get a better understanding of the distribution of the F1-Score values per learners after applying the parameters obtained from tuning, a Repeated K-Fold validation was run on the dataset. In this case, five rounds of 5-fold Cross validation technique was used to obtain the 25 data points which was later fed to Scott-Knott Test. The results of the 5x5 Cross Validation are shown in Fig 7. It can be observed that CART and KNN are much more similar when compared to CART and Random Forests or KNN and Random Forest.

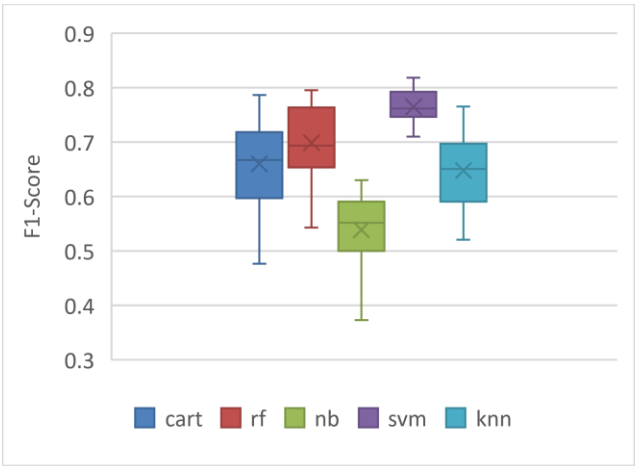


Figure 8. Results of 5x5 cross validation of Lucene (F1-scores)

Scott-Knott Test was run (using `stats.py`) on the results of 5x5 cross validation. The resultant output and the ranks assigned by the test to the learners are shown in Fig 8. The ranks are also tabulated in Table 6. The magnitude of the rank of the learner is directly proportional to the performance of the learner for the given dataset and goal under consideration. Hence Fig 1 can be modified to account for the weights of the individual learners in the ensemble. The modified figure along with the weights is shown in Fig 9. As mentioned before, all the figures and tables shown in this section are specific to the dataset Lucene and optimized for F1-Score as the goal. If the dataset or the goal is changed, the corresponding tuning parameters, ranks and eventually weights will differ. Upon assigning the weights to the individual learners, the resultant ensemble was tuned, and the corresponding performance was observed. The results of the tuning (for F1 score) with Lucene dataset is shown in Fig 10.

Table 6. Scott-Knot ranks for learners (Lucene, F1)

Learner	Rank
Naïve Bayes	1
KNN	2
CART	2
Random Forest	3
SVM	4

	CART		Random Forest		Naive Bayes		SVM		KNN		Ensemble (TS4a)		Ensemble with Tuned Learners (TS4b)	
	Untuned	Tuned	Untuned	Tuned	Untuned	Tuned	Untuned	Tuned	Untuned	Tuned	Untuned	Tuned	Untuned	Tuned
ivy	0.311927	0.210526	0.270833	**0.380952**	0.352941	0.352941	0.034483	0.035088	0.235294	0.2375	0.289157	0.341463	0.289157	0.298851
jedit1	0.562874	0.593023	0.596273	0.590361	0.596273	0.596273	0.600858	**0.606061**	0.559633	0.586207	0.596273	0.590361	0.596273	0.583851
jedit2	0.379747	0.391608	0.476923	**0.506494**	0.46875	0.46875	0.301754	0.308244	0.385417	0.409091	0.474074	0.5	0.474074	0.492537
jedit3	0.095238	0.115702	**0.123457**	0.104478	0.097087	0.097087	0.051661	0.047109	0.062176	0.07619	0.08547	0.113636	0.08547	0.10687
lucene	0.605263	0.666667	0.648101	0.674033	0.529801	0.529801	0.699571	**0.732283**	0.635359	0.635359	0.663342	0.688946	0.663342	0.711688
velocity	0.54955	0.577406	0.564103	0.577406	0.359375	0.359375	**0.588235**	0.584362	0.502463	0.508929	0.568966	0.554688	0.568966	0.561983
xalan1	0.605898	**0.703963**	0.645489	0.541237	0.45812	0.45812	0.675	0.659432	0.603774	0.616092	0.612821	0.687793	0.612821	0.662665
xalan2	0.486942	0.738246	0.574148	0.726241	0.386343	0.386343	**0.751043**	0.748432	0.702312	0.645551	0.686174	0.717143	0.686174	0.701374
xerces	0.31028	0.5	0.322936	0.580247	0.312381	0.312381	**0.606061**	0.459459	0.531882	0.460033	0.340037	0.401421	0.340037	0.444079

Table 8. F1-Score results of all datasets

	CART		Random Forest		Naive Bayes		SVM		KNN		Ensemble (TS4a)		Ensemble with Tuned Learners (TS4b)	
	Untuned	Tuned	Untuned	Tuned	Untuned	Tuned	Untuned	Tuned	Untuned	Tuned	Untuned	Tuned	Untuned	Tuned
ivy	0.786932	0.761364	0.801136	0.775568	0.8125	0.8125	**0.840909**	0.821023	0.630682	0.775568	0.823864	0.832386	0.823864	0.838068
jedit1	0.766026	0.753205	0.791667	0.785256	0.791667	0.791667	0.701923	0.714744	0.692308	0.769231	0.798077	**0.810897**	0.798077	0.804487
jedit2	0.73297	0.792916	0.814714	0.80654	0.814714	0.814714	0.457766	0.476839	0.678474	0.72752	0.743869	**0.820163**	0.743869	0.782016
jedit3	0.806911	0.603659	**0.855691**	0.804878	0.810976	0.810976	0.477642	0.487805	0.632114	0.802846	0.77439	0.845528	0.77439	0.739837
lucene	0.558824	0.638235	0.591176	0.658824	0.582353	0.582353	0.588235	0.573529	0.611765	0.585294	0.605882	**0.661765**	0.605882	0.635294
velocity	0.563319	0.366812	0.554585	0.550218	**0.641921**	0.641921	0.572052	0.558952	0.558952	0.519651	0.593886	0.585153	0.593886	0.554585
xalan1	0.667797	0.691525	0.684746	**0.703955**	0.641808	0.641808	0.647458	0.538983	0.620339	0.620339	0.687006	0.691525	0.687006	0.696045
xalan2	0.330033	0.452145	0.409241	0.566557	0.248625	0.248625	0.606161	**0.608361**	0.546755	0.482948	0.415842	0.442244	0.415842	0.515952
xerces	0.372449	**0.630952**	0.372449	0.537415	0.386054	0.386054	0.513605	0.270408	0.488095	0.447279	0.396259	0.377551	0.396259	0.433673

Table 7. Accuracy results of all datasets

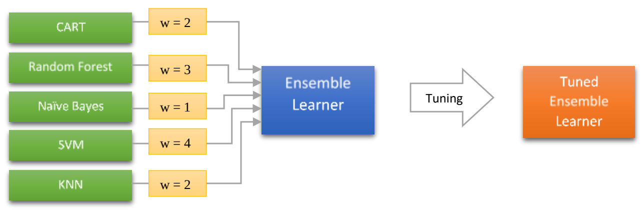
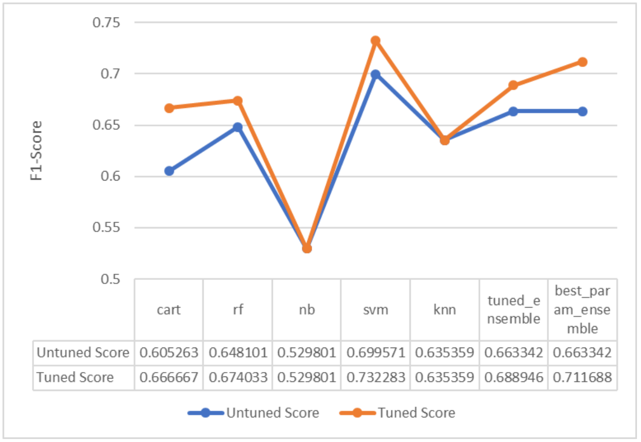


Figure 9. Adding weights to learners before tuning



	cart	rf	nb	svm	knn	tuned_ensemble	best_param_ensemble
Untuned Score	0.605263	0.648101	0.529801	0.699571	0.635359	0.663342	0.663342
Tuned Score	0.666667	0.674033	0.529801	0.732283	0.635359	0.688946	0.711688

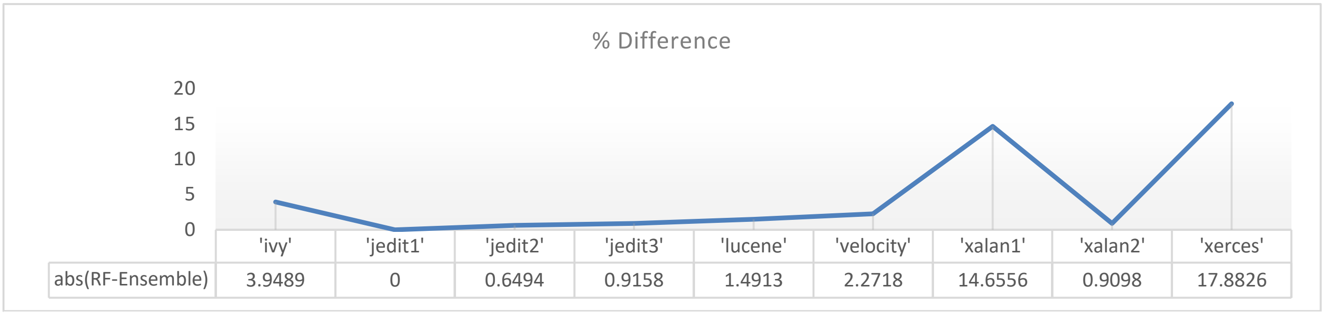
Figure 10. F1-score comparison of all the learners for Lucene

4 RESULTS

Performance comparison was performed using **F1-scores** and **Accuracy** for individual learners and ensembles. Table 7 provides the complete list of the performance of learners where the performance goal is F1-Score. The best value for the performance parameter is highlighted in bold. The two additional columns "**Ensemble (TS4a)**" and "**Ensemble with Tuned Learners (TS4b)**" are the main results. As explained earlier, **TS4a** (Fig 1) corresponds to the case where the ensemble is tuned as a whole and **TS4b** (Fig 2) corresponds to the test case where tuned learners from TS2 are used to form the ensemble as shown in Fig 1 and Fig 2 respectively. Only for 3 datasets (lucene, velocity and xerces) TS4b outperformed TS4a (underlined in Table 7). Table 8 shows a similar trend for Accuracy where only 4 (ivy, xalan1, xalan2 and xerces) of the 9 datasets have been shown to have TS4b performing better than TS4a.

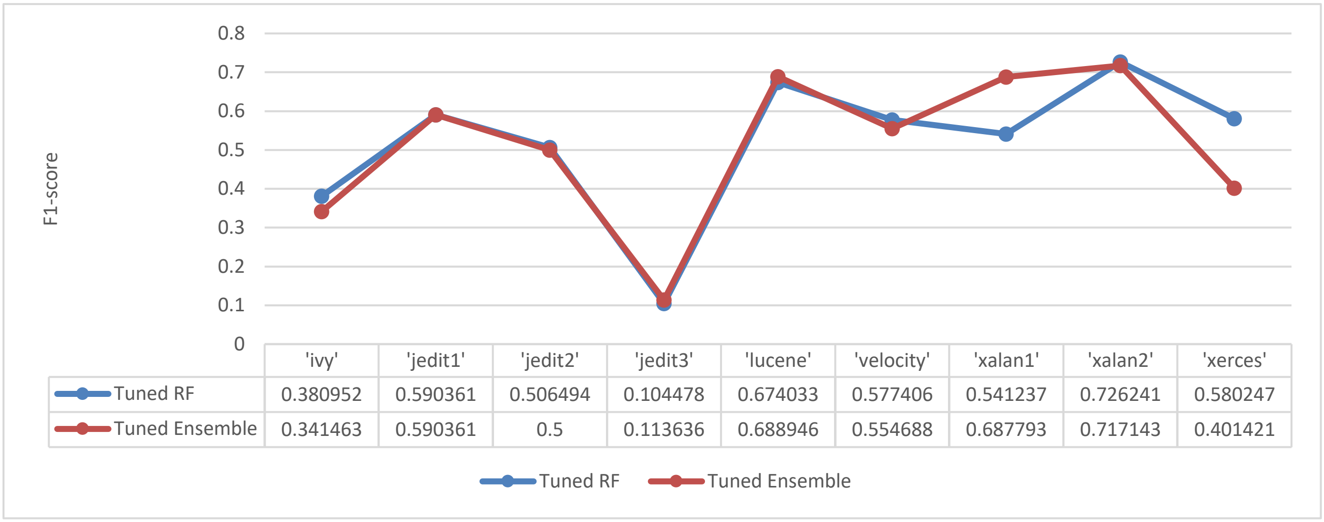
RQ1: *Does the application of ensemble methods (**TS3**) on the data learning algorithms always result in better performance than defaults (**TS1**)?*

Application of ensemble methods (TS4a and TS4b) for F1-score was unable to result in better performance compared to individual learners. As seen in Table 7's Ensemble (TS4a) Untuned column, ensemble did not provide the best performance (highlighted in bold) compared to other learners for any dataset. Upon closer, inspection the ensemble's performance is very close to the second worst learner in the ensemble. More specifically, the ensemble's untuned performance score is decided by the learners with Scott-Knott ranks of 2. A similar behavior was observed with accuracy, as shown in Table 8, where the performance of the ensemble was only slightly better than two of the poorly performing learners. Hence application of ensemble did not result in any performance benefits when compared to their untuned counterparts of individual learners.



	'ivy'	'jedit1'	'jedit2'	'jedit3'	'lucene'	'velocity'	'xalan1'	'xalan2'	'xerces'
abs(RF-Ensemble)	3.9489	0	0.6494	0.9158	1.4913	2.2718	14.6556	0.9098	17.8826

Figure 12. Absolute difference between F1-scores of RF and Ensemble



	'ivy'	'jedit1'	'jedit2'	'jedit3'	'lucene'	'velocity'	'xalan1'	'xalan2'	'xerces'
Tuned RF	0.380952	0.590361	0.506494	0.104478	0.674033	0.577406	0.541237	0.726241	0.580247
Tuned Ensemble	0.341463	0.590361	0.5	0.113636	0.688946	0.554688	0.687793	0.717143	0.401421

Figure 11. F1-score comparison between Tuned RF and Tuned Ensemble

RQ2: *Does tuning the parameters for learners (**TS2**) always result in better performance compared to defaults (**TS1**)?*

For F1-score (Table 7), single learners have shown major improvement because of tuning. All the (learner, dataset) tuples for which tuned performance is better than the untuned performance have been highlighted in orange. Specifically, CART, Random Forest and Ensemble seemed to have greatly benefitted by tuning. Hence the application of tuning on data learning algorithms might not *always* result in best performance. But in most of the cases, the performance is improved. For F1-score, tuning positively impacted the performance of 38 out of 54 cases. That is, nearly 70% of the test cases benefitted from tuning. Similar analysis for Accuracy (Table 8), shows that ensemble learner greatly benefitted from tuning (both TS4a and TS4b). Though, the number of cases for which tuning positively impacted the accuracy is less than that of F1-score. Only 31 of the 54 testcases (approximately 57%) benefitted from tuning.

RQ3: *Does the combination of ensemble methods and parameter tuning (**TS4**) always yield better results compared to direct application of ensemble methods or parameter tuning?*

From Table 7, though tuning the ensemble improved the performance for seven out of nine datasets, it was still not the best performer. The improvements generated by tuning is still not enough to exceed the performance of the best learner among individual learners. The performance seemed to be slightly below than the best or second-best performer when compared to the tuned counterparts of individual learners for the same dataset. Ensemble with tuned learners performed better with only 3 out of 9 datasets. But in most cases TS4b was very close in performance when compared to TS4a. When compared to the tuning time required for TS4a (shown in Fig 13) and TS4b, TS4b will be significantly faster because the time taken by TS4b is less. It can be approximately computed by computing the sum of all the times consumed by the individual learners.

$$Time(TS4b) \approx \sum Time(Learner_i)$$

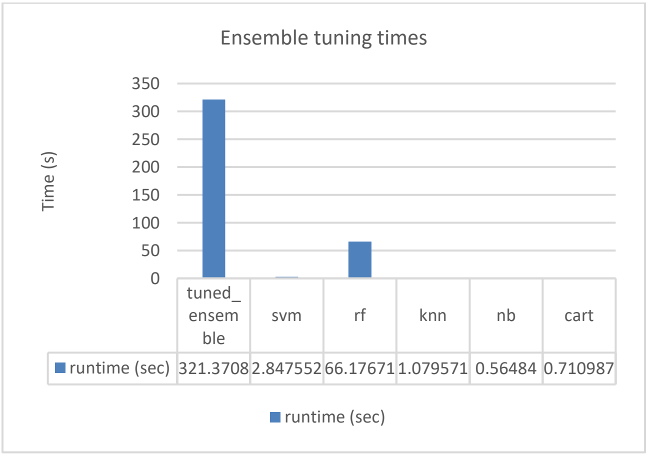
From Figure 10,

$$Time(TS4a) = 321.37\ secs$$

$$Time(TS4b) \approx \sum Time(Learner_i)$$

$$= (2.84 + 66.18 + 1.08 + 0.56 + 0.71) = 71.37\ secs$$

Hence, TS4a consumed roughly 4.5 times more time to tune than TS4b. Hence, creating an ensemble with tuned learner (TS4b) provides almost the same performance as a tuned ensemble (TS4a) and the computation overhead is significantly lower than TS4a. Given that the ensemble was unable to perform better than the best individual learner, it would be beneficial to use TS4b.



	tuned_ensemble	svm	rf	knn	nb	cart
■ runtime (sec)	321.3708	2.847552	66.17671	1.079571	0.56484	0.710987

Figure 13. Time taken for tuning (Lucene, F1)

Upon examining Accuracy results from Table 8, it can be seen that Tuned ensemble was better than the best individual learner for three datasets (jedit1, jedit2 and lucene). In most of the cases, the tuned ensemble has performance in $\pm 2\%$ range of the best individual learner. Hence, unlike in the case of F1-score, tuning the ensemble was greatly beneficial for accuracy. But TS4b is a close contender for TS4a in only half of the cases. Though TS4b, consumes significantly less time compared to TS4a, the performance compromise is not worthwhile.

4.1 Ensemble vs Random Forest

Upon close examination of Table 7, it was observed that the F1-scores generated by a tuned ensemble and the F1-score generated by a tuned Random Forest were very close in magnitude. As shown in Fig 11, the F1-scores almost overlap each other. Fig 12 examines the degree of the closeness by evaluating the absolute difference between F1-scores of Tuned Random Forest and Tuned Ensemble learner. It can be seen that for majority of the cases, the difference is under 4%. This lead to the speculation about the similarities between Random Forests and Ensembles. Tuning an ensemble works by computing the scores of different candidates in the frontier, where a single candidate will have the values for the parameter of all the constituent algorithms. Given the random nature in which crossover occurs in Differential Evolution and candidate generation, the process closely mirrors what happens in a Random Forest where the different learners are created with subsets of the dataset. Hence, the similarity of the process can be accounted as one of the reason for the similarity in their performance.

5 THREATS TO VALIDITY

i. The experiment was performed in a tightly controlled environment where the datasets chosen were all based on CK metrics [2] and the learners were chosen in such a way that there was a heterogenous mixture of learners in the system. Its quite possible that the performance of one learner might be coupled to the performance of another learner (during ensemble tuning) even for learners which belong to different Scott-Knott ranks because of the way candidates are chosen for the frontier in Differential Evolution.

ii. The weights to the learners are directly assigned from their Scott-Knott ranks. In an ensemble of 5 learners, if there are 4 bad learners and 1 good learner, the impact of the poor learners is more pronounced in the ensemble which leads to decline in the performance. Hence the weights assigned to the learners in the ensemble must be multiplied by a factor to avoid such cases. The initial idea was to use exponentiation of the ranks as weights. That is, $weight = 2^{rank}$, but this would always cause the best learner to dominate in the results without allowing other learners to even participate. Eventually, it was decided to allow a linear dependency between the ranks and weights.

iii. Currently the variant of Differential evolution used is **DE/rand/1** (according to Storn's convention [5]). The convergence and the performance could have been better if a different variant such as **DE/best/2** was used. The idea was to avoid Differential Evolution to cause the optimization to result in local maxima instead of global maxima.

iv. The reason that Random Forest's performance was close in magnitude to that of Tuned ensemble could be attributed to the fact that in most of the cases RF is the best learner or second-best learner (according to Scott-Knott ranks). Hence, it's hard to ignore the fact that RF might have a larger role in the ensemble than anticipated which could have caused the Tuned Ensemble's performance to be nearly equal to that of Random Forest.

6 CONCLUSION

Results for **RQ3** show that a tuned ensemble always underperforms when compared to the best individual learner. But with few datasets (tuned for accuracy) the results were indeed better. But tuning an ensemble is much more time consuming that tuning an individual learner (Fig 13). As shown in the earlier sections, the performance of Tuned Ensemble is almost mirrored by Random forest and Ensemble with tuned Learners (TS4b). Hence it is suggestible to tune Random Forest in place of Ensemble with little to no compromise in performance.

7 FUTURE WORK

i. To increase the scope of the project, the experiment will be repeated with datasets which do not use CK metrics (like NASA corpus).

ii. Test Differential Evolution with other variants and verify if there is a performance improvement.

iii. Repeat the experiment with more number of learners and allow different set of learners in each ensemble.

iv. Tune (data) ensemble learners like Bagging and Boosting and compare their performance with individual learners and algorithm ensembles.

v. Verify if other (data) ensemble learners like Boosting or Bagging show similar performance like Random Forest.

8 CODE AND ALGORITHM FLOW

1. Fetches 4 versions of the data namely i) `train`, ii) `tune`, iii) `test`, iv) `merged`. Function of the individual versions are explained in the report.

2. SMOTE is applied on train and merged. None of the test datasets are SMOTEd. SMOTE is performed by `preprocess()` in `preprocess.py`. It uses SMOTE from imblearn package.

3. All the learners are tuned using Differential Evolution from `DiffentialEvolutionTuner()` in `DE.py`.

4. The output of the learners are written to the file `../results/<goal>/<dataset>_results.csv`. For example, if run.py was called for Lucene dataset and Accuracy as the goal, the file would be named `../results/f1/lucene_results.csv`. The contents of the file are in the following format:

   ```
   Learner | Best parameters (computed by the tuner) |
   Untuned Score | Tuned Score
   ```

5. Applying the "Best Parameters" obtained from the tuner, the learners will perform a 5x5 Cross validation on the dataset. Their results are stored in `../results/<goal>/<dataset>.kout`

6. Scott-Knott Test will be run on the `*.kout` file to obtain the ranks which will be used as weights by the ensemble learner.

7. With the weights computed and the learners, `VotingClassifier` is created whose parameters are the union of all the parameters of all the constituent learners.

8. Best Parameters and the corresponding scores of the ensemble are written to `../results/<goal>/<dataset>_results.csv`.

9. A new ensemble is created by applying the best params to individual learners and the corresponding scores are captured and appended to the file `../results/<goal>/<dataset>_results.csv`.

REFERENCES

[1] Tuning for Software Analytics: is it Really Necessary? Wei Fu, Tim Menzies, Xipeng Shen

[2] A metrics suite for Object Oriented Design. Chidamber and Kermer. *IEEE transactions on Software Engineering (Volume 20, No 6, June 1994)*

[3] Less is More: Minimizing Code Reorganization using XTREE. Rahul Krishnaa, Tim Menzies, Lucas Layman.

[4] Effect of Feature Selection, SMOTE and under Sampling on Class Imbalance Classification. Nadeem Qazi, Kamran Raza. *2012 UKSim 14th International Conference on Computer Modelling and Simulation*

[5] Minimizing the real functions of the ICEC'96 contest by differential evolution. R. Storn, K. Price. *Proceedings of IEEE International Conference on Evolutionary Computation*

[6] Danijel Radjenović, Marjan Heričkob, Richard Torkarcd and AlešŽivkovič. Software fault prediction metrics: A systematic literature review. *Information and Software Technology, Volume 55, Issue 8, August 2013, Pages 1397-1418*

[7] A. Günes Koru, Hongfang Liu. An investigation of the effect of module size on defect prediction using static measures. *PROMISE '05 Proceedings of the 2005 workshop on Predictor models in software engineering.*

[8] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, Kenichi Matsumoto. Automated Parameter Optimization of Classification Y. Jiang,

[9] B. Cukic, and T. Menzies. Can Data Transformation Help in the Detection of Fault-prone Modules? *In Proceedings of the workshop on Defects in Large Software Systems.*

[10] Tibor Gyimo´thy, Rudolf Ferenc, and Istva´n Siket. *Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction.* IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 31, NO. 10, OCTOBER 2005

[11] Precision – Recall (scikit-learn) http://scikit-learn.org/stable/auto_examples/model_selection/plot_precision_recall.html

[12] T. Menzies, J. Greenwald, A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering, 33 (1) (2007), pp. 2-13studies.* Expert Systems with Applications. Volume 36, Issue