Prediction of who should refactor the code Phase 3 - Explorations & Limitations

S. BENDAPUDI, R. KIPP, T. OLATUNBOSUN, and M. SZCZEPANIAK

1 INTRODUCTION

Software quality has been evolving beyond fixing bugs towards reusable, flexible, understandable, functional, extensible and effective code. Ideally, these attributes arise from good design prior to actual implementation. However, good design principles are often skipped due to the lack of developer awareness as well as schedule constraints. Modern software refactoring practices are of interest for cost, schedule, productivity, and quality reasons.

Large projects involve developers with different skill levels. This leads to optimizing the use of developers' strengths, consolidating efforts efficiently, and ensuring a positive value proposition of cost versus technical debt. It is assumed that refactoring efforts will improvement code quality in the dimensions mentioned above. However, as organizations values the impact of refactoring differently, implementation is not consistent and highlights a trade-off between short-term development costs versus long-term quality costs.

Organizations that acknowledge the importance of refactoring face the subsequent execution challenge of assignment. Incorrect or ad hoc assignment of refactoring tasks to developers may result in not realizing the intended benefits by mismatching the task to the developers' skill sets and experience level. Manual assignment is time-consuming due to the volume of requests. It is also organizationally challenging because developers and project leaders are constantly either reassigned to different project or leave the company. This motivates the need for automating the refactor task assignment. For example, a good automated solution would take relevant and available inputs such as the type of refactoring to be done and the file to be refactored and output the developer or group of developers that can best do the task.

The objective of this project is to develop a set of classifier models where the author or group of authors is the predicted response when deciding who should refactor, based on the information contained in a request. We aim to answer the following research questions:

RQ1 – Which of the three classification model tends to give the lowest classification error?
RQ2 – What is the trade-off in overall model performance if improved performance of the low committer group is a priority?
RQ3 – Which features are most important in predicting which developer is best suited for a given refactoring request?

The overall approach is as shown in Figure 1.

2 STUDY DESIGN

The data source file for this project is *project3-authors.csv* [2]. This data contains undated commit logs from three open source projects with the following nine fields:

(1) **Name** – Project name: *adangel$pmd*, *hibernate$hibernate-validator* and *eclipse$bpmn2-modeler*
(2) **CommitId** – This is a 40 character commit hash for each refactoring commit
(3) **RefactoringType** – Description of the refactoring type that took place. There are 27 types ranging from "Move Class" to "Move And Rename Attribute"

Authors' address: S. Bendapudi, sb2426@rit.edu; R. Kipp, rw9493@rit.edu; T. Olatunbosun, tao5634@rit.edu; M. Szczepaniak, mds9178@rit.edu.

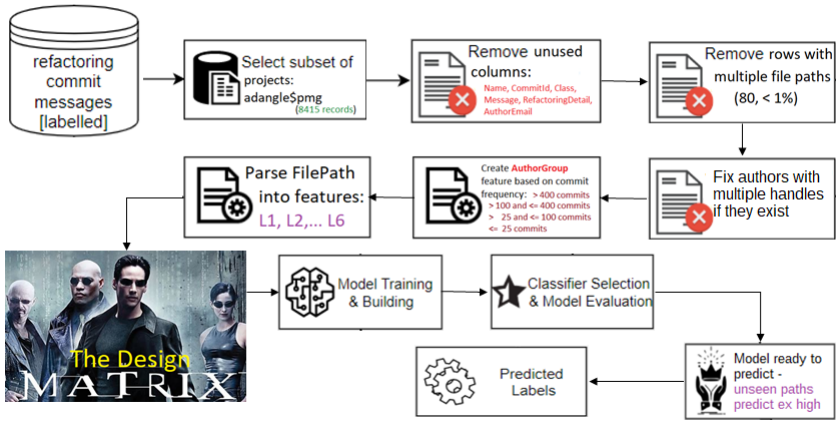

Fig. 1. Overall approach

(4) **FilePath** – The full file path of the file where refactoring changes were made.
(5) **Class** – The class where refactoring changes were made.
(6) **RefactoringDetail** – Description of the refactoring change that was made.
(7) **Message** – The commit message corresponding to the checked in changes.
(8) **AuthorName** – Name of the person who made the commit. There were 77 unique names in this *adangel* project.
(9) **AuthorEmail** – Email of the person who made the commit.

Our data set was aggregated from three open source projects. We built our initial model using the *adangel PMD* project to determine proof of concept. Because the result of the initial model indicated that it could do a reasonably good job of predicting which group of authors made a particular refactoring commit, we applied a similar process to the one described in Figure 1. to the *hibernate validator* project as well and compared the results.

3 EXPERIMENTS

3.1 The data preparation

While the data describes the entire refactoring event, this project is limited to predicting the author. With this objective we can only use data that can be assumed to be available prior to the refactoring. To accomplish this we removed multiple columns, leaving only the file location and type of refactoring to be performed. We assume that these two features can be reasonably expected to be received from a code smell or defect detecting software.

To answer our first research question, we experimented with three different model types and three different sets of features. We tried every combination of models and data representations to see what combination produces the best results through a 10-fold cross validation. Finally, we selected the model that performs best and retrained it on another open source project (*hibernate validator*) to determine if the same approach would work as well as on the *adangel PMD* project.

The *project3-authors.csv* file was processed for analysis by first being read into a python Jupyter notebook where the following data preparation steps were performed:

(1) The file was filtered to extract data for just the *adangel$pmd* project.
(2) The **Name**, **CommitId**, **Class**, **Message** and **AuthorEmail** columns were removed leaving the remaining 4: **RefactoringType**, **FilePath** and **AuthorName**

Table 1. Refactoring Types

Change Package	Inline Variable	Pull Up Method
Extract And Move Method	Move And Rename Attribute	Push Down Attribute
Extract Class	Move And Rename Class	Push Down Method
Extract Interface	Move Attribute	Rename Attribute
Extract Method	Move Class	Rename Class
Extract Subclass	Move Method	Rename Method
Extract Superclass	Move Source Folder	Rename Parameter
Extract Variable	Parameterize Variable	Rename Variable
Inline Method	Pull Up Attribute	Replace Variable With Attribute

Table 2. Parsed columns of example filepath

L1	L2	L3	L4	L5	L6
pmd	src	com	infoether	pmd	ast/ASTAdditiveExpression.java

(3) Rows with more than one file path in the **FilePath** column were removed. There were 80 such rows representing <1 % of the data.

For the *hibernate validator* project, an additional data cleaning step was required. This additional step was required because one of the commit authors *marko bekhta* was listed twice using slightly different identifiers *marko-bekhta* and *marko.bekhta*, and another *Davide D'Alto* had an apostrophe that was causing issues with the model builder. This step was simply to replace the hyphen and the dot by a space for every instance of *marko bekhta* in the data and to remove the apostrophe from every instance of *Davide D'Alto*.

After these preparation steps, we evaluated two responses: the first was the **AuthorName** which consists of 77 distinct values for the *adangel PMD* project and 23 for the *hibernate validator* project. However, the distribution of authors was found to be extremely skewed for both project. Figure 2 shows this distribution for the *adangel PMD* project. The second was **AuthorGroup** which grouped the authors into four categories.

After experimenting with three sets of features DS1, DS2 and DS3, we settled on a design matrix comprised of the following 7 predictors: refactoring type **RefactoringType** which is a categorical variable with 27 levels described in Table 1 and six levels of the file path which were created by splitting the file path into six separate features representing each part of the path starting at the top. The last level (L6) was whatever remained. For example, the single file path: *pmd/src/com/infoether/pmd/ast/ASTAdditiveExpression.java* was parsed into the columns shown in Table 2. This gave us the following 7 input parameters:

(1) RefactoringType
(2) L1 - Level 1 of the path
(3) L2 - Level 2 of the path
(4) L3 - Level 3 of the path
(5) L4 - Level 4 of the path
(6) L5 - Level 5 of the path
(7) L6 - Remainder of the path, including object name

Fig. 2. Number of commits by developer - *adangel* project

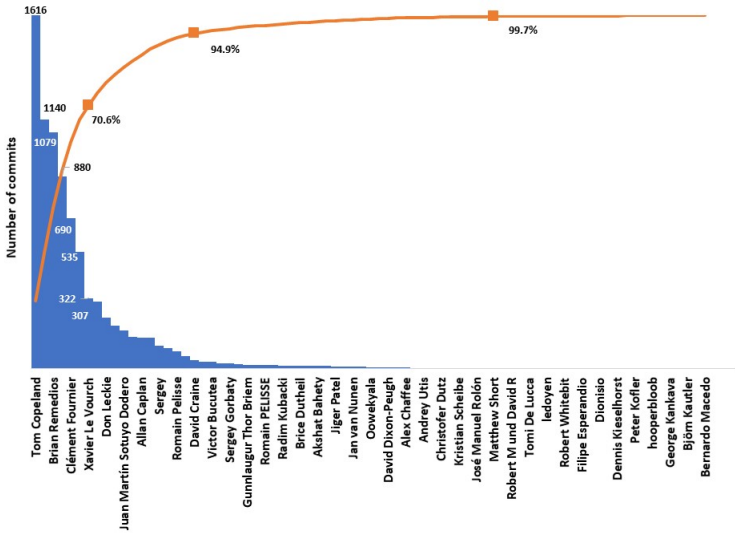

Table 3. Grouping of authors by commit counts

# of commits	AuthorGroup	# of committers
>400	extremely high	6
>100 and ≤ 400	high	9
>25 and ≤ 100	medium	8
≤25	low	54

Instead of predicting author name, we chose to change the response to an author grouping **AuthorGroup** which is based on the amount of commits each author has. The bins selected are shown in Table 3.

Choosing the response in this way has two main advantages. First, instead of assuming constant competency hierarchy over all the developers, a looser assumption of competency levels between groups needs to be made. Second, by classifying a group, a scheduler has the flexibility to assign a refactoring task to whichever developers have availability with the group.

While this design matrix has many advantages, it has a known vulnerability because in that it will not handle requests on an input file that the model did not see in training. The most practical way to handle this implementation in practice would be to assign such tasks to the *extremely high* committer group as shown in Figure 1. By doing this, one would eliminate the chance of incurring a *Redoing cost* which is explained in section 4.3.

3.2 The Classifier Models

Each of the three datasets above were imported in to Weka [1] and three classifier models were applied to each: Naïve-Bayes, J48 Decision Tree and Random Forest. The models were trained using 10-fold cross validation. The J48 model was fit using a default confidence factor of 0.25, a parameter to help control the extent of pruning - the lower the confidence factor, the more extensive the

Table 4. Classifier models for the adangel project

	DS1			DS2			DS3		
	NB[1]	J48[2]	RF[3]	NB	J48	RF	NB	J48	RF
Accuracy %	85.8	87.7	89.4	73.7	81.2	84.5	82.3	84.3	88.2
RMSE %	19.6	17.6	16.9	25.6	20.2	19.8	26.1	25.2	20.7

Table 5. Summary statistics of the *adangel* classifier models

	Naïve-Bayes		J48 Decision Tree		Random Forest	
Correctly Classified Instances	6922	82%	7094	84.30%	7421	88.19%
Incorrectly Classified Instances	1493	18%	1321	15.70%	994	11.81%
Kappa statistic	0.610		0.595		0.713	
Mean absolute error	0.132		0.126		0.097	
Root mean squared error	0.261		0.252		0.207	
Relative absolute error	57.935%		55.273%		42.688%	
Root relative squared error	77.44%		74.91%		61.49%	
Total Number of Instances	8415		8415		8415	

Table 6. Confusion matrices for the *adangel* classifier models

Naïve-Bayes				J48 Decision Tree				Random Forest					
a	b	c	d	a	b	c	d	a	b	c	d	<–	classified as
5312	380	58	190	5832	87	14	7	5815	76	19	30		a = extremely high
446	1206	42	78	794	960	14	4	605	1141	13	13		b = high
97	15	236	40	106	30	231	21	68	27	281	12		c = medium
118	13	16	168	228	3	13	71	115	9	7	184		d = low

pruning. The Random Forest model was fit with the default bag size percent of 100. Table 4 shows the results of each of these models for the three data sets we considered.

3.3 Results and Observations

Across the three datasets, the Random Forest model appears to provide not only the best classifier accuracy as shown in Table 4, but the area under the ROC curve (AUC) and the PR characteristics also look better than the other two models as well. While DS3 has higher RMSE values, the bucketizing of author names makes it more scalable to other data sets which was another reason we selected it moving forward.

4 MODEL PERFORMANCE

4.1 Types of Errors

Because we are not doing binary classification, there are multiple ways to generate false positive (FP) errors. For example, a FP for classifying 'extremely high' instances would be classifying an actual 'low', 'medium' or 'high' as 'extremely high'. Similarly, a FP for classifying 'high' instances would be classifying an actual 'low', 'medium' or 'extremely high' as 'high'. The same pattern applies to the 'medium' and 'low' classes.

Fig. 3. ROC (left) and Precision Recall curves (right) for 3 models evaluated on *adangel project*

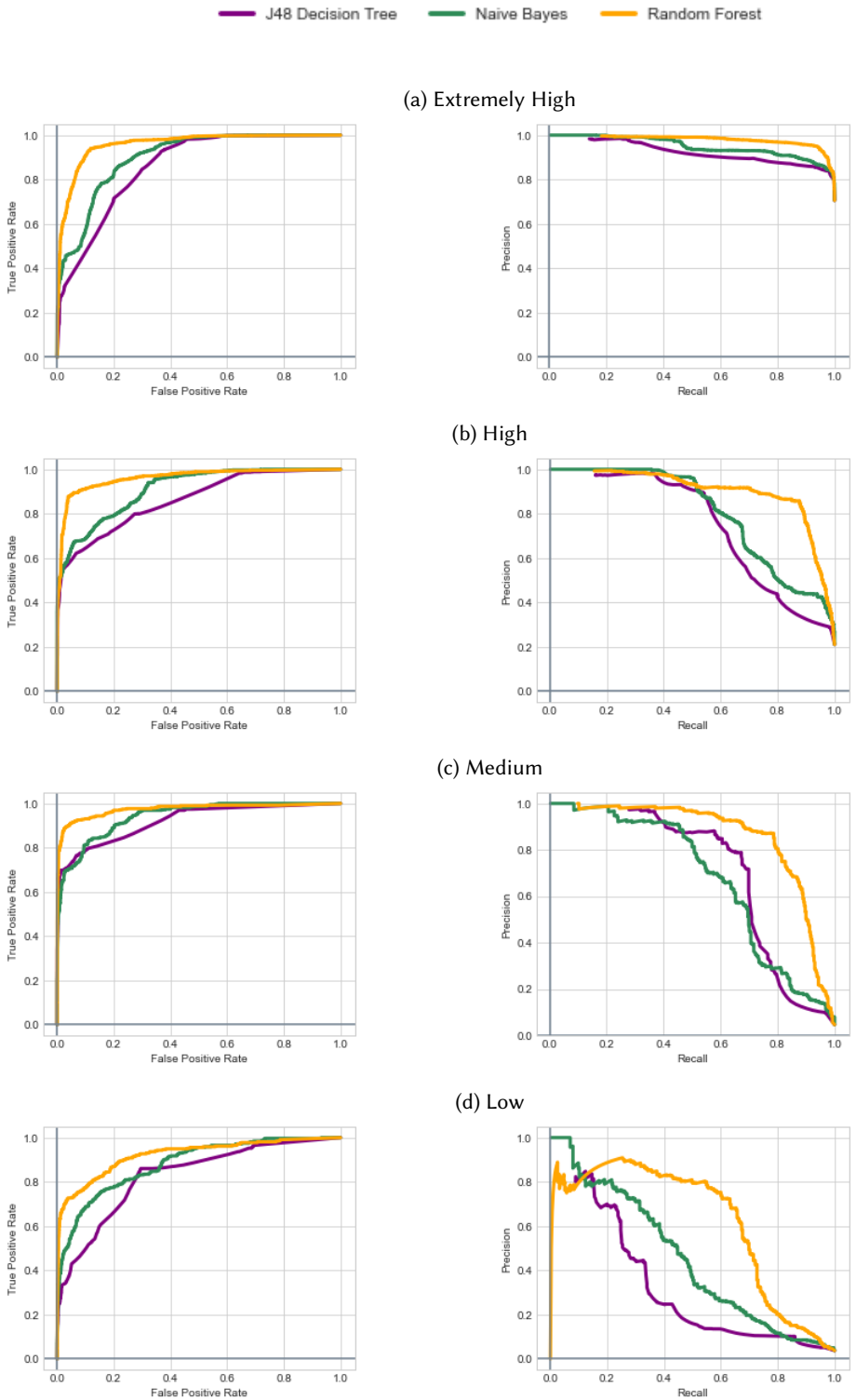

Fig. 4. ROC (left) and Precision Recall curves (right) Random Forest on Two Projects

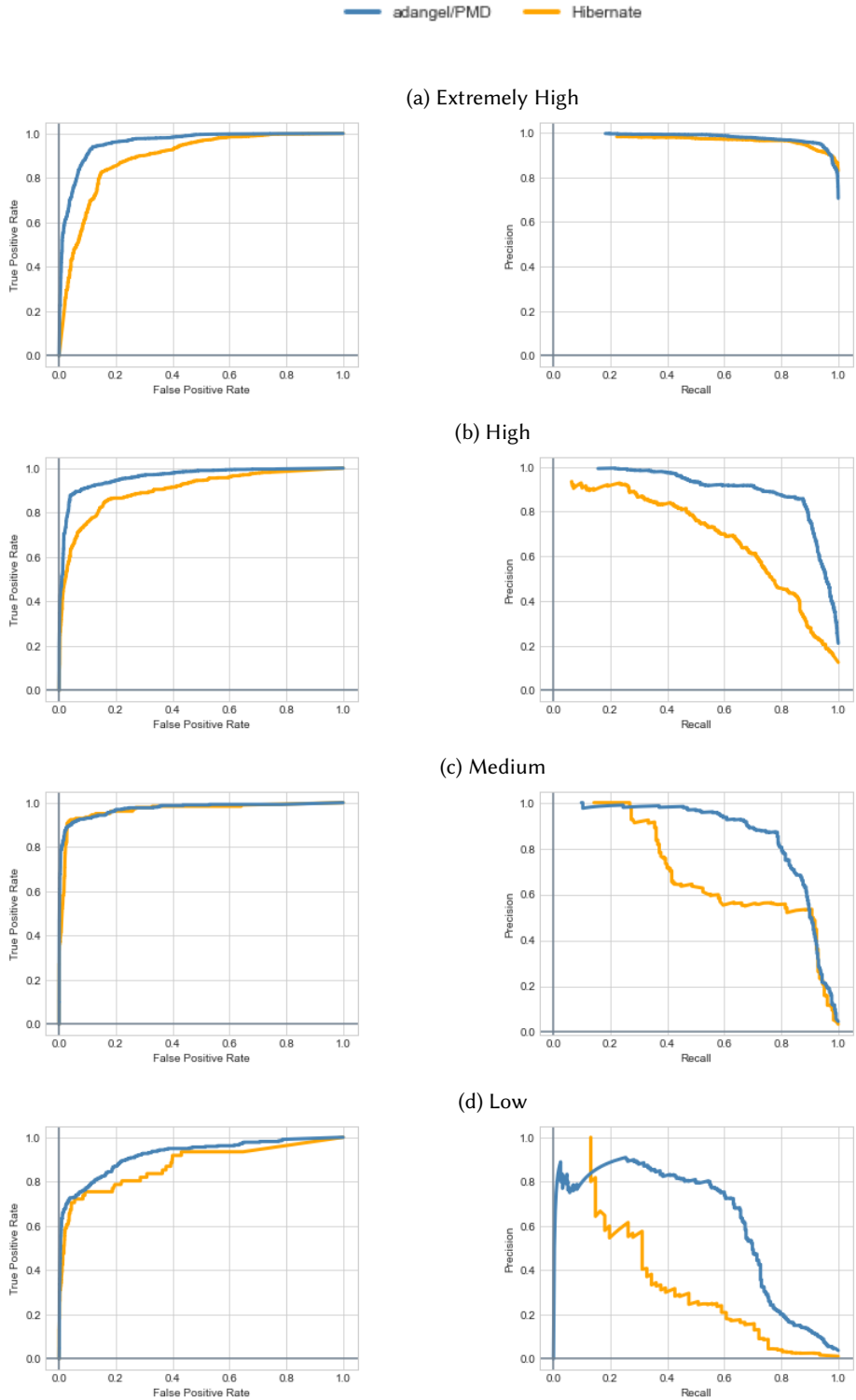

Table 7. Confusion matrices for the hibernate and adangel Random Forest classifiers

Random Forest - hibernate				Random Forest - adangel					
a	b	c	d	a	b	c	d	<–	classified as
4279	116	11	7	5815	76	19	30		a = extremely high
311	356	0	0	605	1141	13	13		b = high
116	0	66	1	68	27	281	12		c = medium
50	0	0	11	115	9	7	184		d = low

Table 8. Examples of False Positives and False Negatives for the Low Class on adangel Project

RefactoringType	L6	Actual	Predicted
Extract Method	util/designer/Designer.java	ext. high	low
Move Class	.../...ForNonOverlappingRegions.java	high	low
Rename Parameter	AvailableListModel.java	medium	low
Pull Up Method	.../AvoidFieldNameMatchingMethodName.java	low	ext. high
Rename Method	.../ApexTreeBuilder.java	low	high
Extract Variable	pmd/jedit/PMDJEditPlugin.java	low	medium

Since we have 4 classes, we will have 4 - 1 = 3 ways to generate false negative (FN) as well. For example, a FN for classifying 'extremely high' instances would classifying an actual 'extremely high' as 'low', 'medium' or 'high'. Similarly, a FN for classifying 'high' instances would be classifying an actual 'high' as 'low', 'medium' or 'extremely high'. Again, the same pattern applies to the 'medium' and 'low' classes.

The first 3 rows of Table 8 are examples of FP's and the last 3 rows are examples of FN's all for the *low* class. The first 5 path level predictors L1 through L5 are not shown for brevity. These errors were chosen because mis-classification of the low class is assumed to be the most costly to an organization. There is nothing about these examples that provide any immediate insights into why they were mis-classified. This is because a Random Forest (RF) classifier makes its predictions based on an aggregation of many de-correlated decision trees as depicted in Figure 5.

4.2 Random Forest Workings and its Meta-Parameters

This de-correlation is accomplished by selecting a random subset of features to build each tree. For example, one tree might be built from the L1, L3 and L5 variables shown in **Tree #1** of Figure 5. The number of features used for each tree is a meta-parameter that is typically $\sqrt{n}$ where n is the total number of features. Because this is a well established default, we didn't experiment with different numbers of features per tree. Since the errors each tree makes is not correlated with the other trees, the model learns to make an unbiased estimate for the conditional probability of the class given the predictors.

Two other RF meta parameters are the values for M which is the number of trees created shown as the subscript of N and the depth of each tree in the forest D. In Figure 5, D = 2.

Both of these meta-parameters are important because setting either of them too high can cause the RF to overfit the data. At the other extreme, setting either too low can lower the performance. Defaults for these values for our models were M = 100 and D = 7. The results of changing these values did not appreciably change the results as shown in Table 9, so the defaults were used for the rest of the analysis.

Table 9. Overall Accuracy For RF Meta-Parameter Changes on Hibernate Project

# of Trees	Depth=5	Depth=6	Depth=7
90	88.5424	88.5237	88.5237
100	88.4861	88.8049	88.5049
110	88.4673	88.4861	88.4861

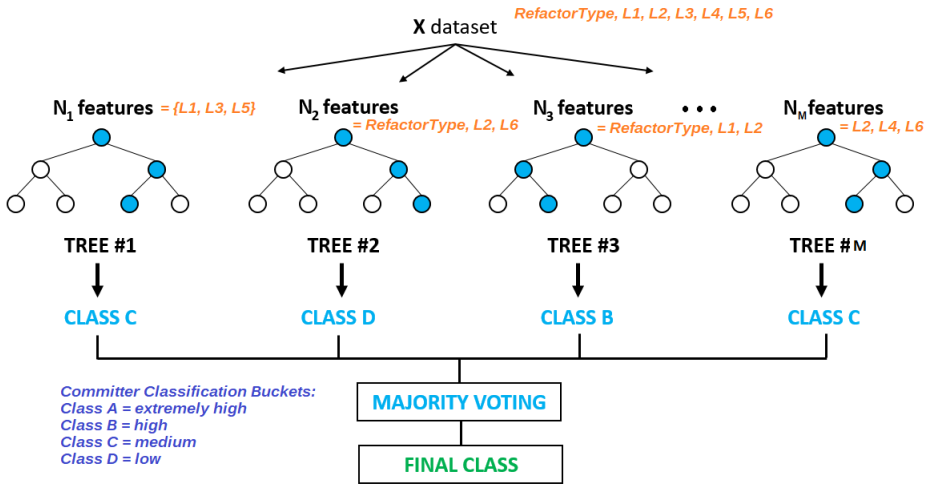

Fig. 5. How a Random Forest Works

4.3 Performance Trade-offs

There are two basic types of errors that can be made in making refactoring assignments. The first we call *Redoing cost* which is the cost of having to redo a task because it was assigned to a low or medium committer but should have been assigned to a high or extremely high committer group developer. The second error we call *Experience cost* which is the cost of assigning a task that could have been done by a low or medium committer but was assigned to a high or extremely high committer.

Our second research question was motivated by the assumption that *Redoing cost* > *Experience cost*. Under this assumption, the PR curves on the right of Figure 4. informs us of the trade-off between *Redoing cost* and *Experience cost* because we want to maximize the proportion of actual positives (recall) of the low group while keeping the proportion of predicted positives (precision) as large as possible. Because determining and using an alternative threshold cut-off to map probabilities of a class given data to a class has been found to be preferable to using alternative sampling techniques such as SMOTE [3], we can determine a reasonable trade-off for *hibernate* appears at a Recall of roughly 0.30 with Precision of approximately 0.6 for the low committer group is shown in Figure 6. The threshold cut-off value at this point is 0.383 which means we should not assign a refactoring task to the low group unless the probability of the low class given the data is at least this value.

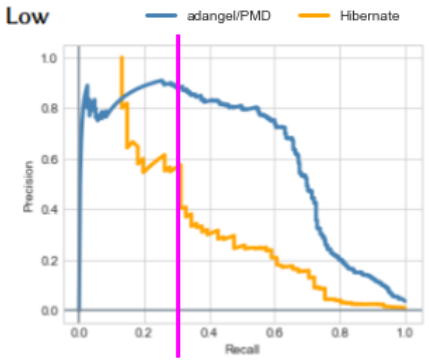

Fig. 6. Low Group Performance Trade-offs

Table 10. Variable Importance by Information Gain for the *adangel* and *hibernate* project models

adangle Project		hibernate Project	
IG Rank	Predictor	IG Rank	Predictor
0.9627	L6	0.54682	L6
0.3198	L1	0.0647	L1
0.1009	L3	0.05525	RefactoringType
0.0982	L5	0.00968	L5
0.0782	L4	0.00752	L3
0.0780	RefactoringType	0.0069	L2
0.0552	L2	0.00581	L4

4.4 Variable Importance

To answer our third research question question, each predictor was assessed in terms of evaluating the worth of an attribute by measuring the information gain with respect to the class. Table 10. shows that **L6** and **L1** are the top two contributors to the model in both projects. **RefactoringType** is a close third for the hibernate project with the fourth most important predictor (**L5**) dropping in importance by almost an order of magnitude. For the adangle project, **RefactoringType** is the sixth most important, however the difference in relative ranking between the third and sixth predictors is small. This provides reasonable evidence that **RefactoringType** can be considered the third most important predictor in general based on looking at two projects.

5 THREAT TO VALIDITY

5.1 Internal Validity

Our first and primary threat to validity arises from our limited feature space and is based on an implied assumption that committers in the different frequency buckets work on different files. For example, if the low committers are working in similar file locations as the other classes (extremely high, high and medium), or even on the same files, the model will have a difficult time distinguishing between the classes.

There is strong evidence that this first threat explains the difference in performance between the *adangle* and *hibernate* projects characterized by PR curves for the low and medium classes shown

Commit Group	low files committed by ex high	Total committed files	proportion of total	P-value, H01: $p_{low,\ ada} = p_{low,\ hib}$ H02: $p_{med,\ ada} = p_{med,\ hib}$
adangle – low	115	315	0.3651	HA1: $p_{low,\ ada} < p_{low,\ hib}$
hibernate – low	43	63	0.6825	3.039e-06
adangle – medium	96	388	0.2474	HA2: $p_{med,\ ada} < p_{med,\ hib}$
hibernate – medium	78	183	0.4262	1.146e-05

Fig. 7. Testing file overlap proportions

in Figure 4. This evidence is presented in Figure 7. which shows the results of a test of proportions of files committed by both the low and extremely high groups and the medium and extremely high groups.

On both the low and medium groups, the hibernate project showed a significantly larger portion of overlap with files committed by the extremely high group. This shows us that a strong negative correlation exits between file overlap and model performance.

5.2 Construct Validity

A second threat is based on another assumption which is that commit frequency is a valid proxy for capability to do a refactoring well. The following example illustrates a situation where this assumption breaks down: A low committer may have low commit counts for a particular project, but may be more capable because they have experience on other projects that our data is not reflecting.

6 SUMMARY AND CONCLUSIONS

The *adangel* and *hibernate* projects data were extracted from the *project3-authors.csv* data file. The data was prepared using three distinct approaches to yield three different sets of features on the *adangel* project. The last approach organized the response (authors) into 4 groups, used a categorical refactoring type as one predictor and parses the file paths predictor into 6 additional categorical predictor variables. This last approach, referred to as DS3 was selected for the remainder of the project and was also applied to the *hibernate* project data.

6.1 RQ1. Which of the three classification model tends to give the lowest classification error?

As part of evaluating the data sets, three classifier models were built from the *adangel* data: Naïve-Bayes, J48 Decision Tree and Random Forest. From a comparison of the data sets and the models performance, the Random Forest classifier resulted in the best combination of classifier accuracy (88.2%), as well as ROC and Precision-Recall characteristics.

A Random Forest classifier was also built using the *hibernate* project data using the DS3 approach. The ROC and PR curves comparing both projects are shown if Figure 4. Models built for both projects show a similar pattern of good performance predicting the *Extremely High* class and degrading as we move to the lower frequency committer classes. This can be explained by the fact that the model has far more samples to learn this class than the other three.

6.2 RQ2. What is the trade-off in overall model performance if improved performance of the low committer group is a priority?

Figure 4. suggest that model performance depends strongly on the kind of project the model is being built for. The **hiberate** project had less samples to train on than the **adangel** project, but there is strong evidence as shown in Figure 7 that the more overlap of different author groups working on the same files there is, the worse the model will perform.

Because the *Extremely High* committers dominate the data as the majority class, a good performance metric to focus on is the **recall** for the *Low* (minority) class. The **recall** measures the proportion of actual positives that are classified correctly. The recall for the *Low* class on *adangel* project was 0.5841 and 0.1803 on the *hibernate* project indicating that *adangel* model did a threefold better job of predicting the *Low* class than the *hibernate* model.

To improve the recall in the low committer group, one needs a sense for the cost in lost precision. An example of determining an alternative threshold probability (0.383) in which to classify a task to the low group was provided for the *hibernate* project. The same procedure can be followed for other projects.

6.3 RQ3. Which features are most important in predicting which developer is best suited for a given refactoring request?

The top 3 predictors which contribute to the model base on information gain are **L6**, **L1** and **RefactoringType** respectively. This result matches our intuition since **L1** and **L6** by themselves would likely categorize most file paths with the middle levels **L2 - L5** providing little additional information. The **RefactoringType** importance also matches our intuition. However, this predictor likely suffers from the same overlap problem that was uncovered with file paths.

6.4 Why Weka?

Weka is an open-source Java-based Machine-learning application with a rich API allowing it to be used in the development of other Java applications [1]. The Weka platform was used to build the models evaluated in this project. Weka was chosen because of its accessibility. Because the GUI version does not require the user to have expertise in any particular language such as R, Python or Java, the tool can be effectively used by non-Data Science professional. While easy to use, it is also quite powerful making it a good choice as a prototyping tool.

6.5 Future Work

By testing the proportion of files worked on by both extremely high group committers versus low and medium group committers, we showed that the difference in performance between projects was strongly correlated to the inverse of the amount of overlap in the files these groups worked on. Model performance will also suffer when there is strong overlap in refactoring types. Future work which quantifies the overlap of files worked on as well as refactoring types over a large number of projects could provide evidence as to whether or not these predictors have general utility in modeling refactoring requests.

REFERENCES

[1] Eibe Frank, Mark A Hall, and Ian H Witten. 2016. Weka. *Data Mining: Practical Machine Learning Tools and Techniques* (2016).

[2] Mohamed W Mkaouer. 2022. DSCI 644 Spring 2022 Project Datasets. (February 2022). https://mycourses.rit.edu/d2l/le/content/960221/viewContent/7969399/View

[3] Ruben van den Goorbergh, Maarten van Smeden, Dirk Timmerman, and Ben Van Calster. 2023. The harm of class imbalance corrections for risk prediction models. (February 2023). https://arxiv.org/pdf/2202.09101