Predicting Refactoring Using Machine Learning

Elizabeth Milne
Department of Computer Science
University of Victoria
emilne@uvic.ca

Sanjay Dutt
Department of Electrical and Computer Engineering
University of Victoria
sanjaydutt@uvic.ca

Abstract

Code refactoring is a technique that changes code structure without affecting its output. Studies show that refactoring code can facilitate collaboration and decrease cost, however automated refactoring detection tools are currently very limited. This project analyzes the effectiveness of using different machine learning models to predict method-level refactoring. The results indicate that all the ML models have high accuracies, but some are considerably better than others.

1 Introduction

1.1 Background

To understand why code refactoring is necessary, it is important to understand what refactoring actually is. Refactoring is a technique that changes the structure of a body of code without changing the results it produces [1]. It is used to simplify existing code which can make it more easily understood by other programmers (or the same programmer later). Refactoring code is a key tool for reducing technical debt that can cause problems later on. There are many benefits to refactoring early and often, such as improving collaboration, lowering the cost of future modifications to the project, and sometimes even decreasing memory or runtime [2].

1.2 Problem Statement

Refactoring code can have positive effects on a body of code's adaptability, maintainability, understandability,reusability, and testability [3]. However, while many automated refactoring tools exist, they have limitations [4]. Their refactoring detection methods often rely on simplistic metric thresholds (such as the number of lines of code in a method) which can result in frequent false positives [4]. In order to fully capture the complexity of the factors involved in identifying refactoring opportunities, it seems a more sophisticated method needs to be used, such as machine learning algorithms.

This project is intended to reimplement the methods used in the paper "The Effectiveness of Supervised Machine Learning Algorithms in Predicting Software Refactoring" by Mauricio Aniche, Erick Maziero, Rafael Durelli and Vinicius H. S. Durelli [4]. This paper explored using machine learning to determine when code refactoring is necessary, and predict what kind of refactoring should be used [4]. Like the authors of this paper, this project will compare the results of using six different ML models (Decision Tree, Random Forest, Naive Bayes, Logistic Regression, Support Vector Machines, and Neural Network) for this problem. However, it has been slightly simplified in that it focuses only on method level refactorings and uses a smaller subset of data.

1.3 Dataset

The dataset used in the aforementioned paper was obtained from three online sources (Apache, F-Droid, and Github) using an automated data collection tool created by the researchers. They inputted

9037 projects (which totalled 8.8 million commits in March 2019) into the tool and checked for 4 types of refactoring: Class-level refactorings, Method-level refactorings, Variable-level refactorings and Non-refactoring instance, each of which have their own features unique to the type of refactoring (for example, "Extract And Move Method" is a feature in the Method-level set) [4].The tool then collected the code metrics from before and after the refactoring was done. After running the tool, the researchers were able to generate 3,063551 pieces of data [4]. Unfortunately, we did not have the same resources and time to retrieve these same results, so we chose to simplify the problem by focusing on the Method-level refactorings from 100 of the 9037 projects. This left us with 348,058 pieces of data. It's also worth nothing that since the original data was collected in March 2019, more commits to the projects may have been completed since then providing our models with some slightly different data points.

Since the data tool returned the refactored and non-refactored instances as separate SQL databases, we also joined them and labelled the non-refactored code as 0 and refactored code as 1.

Table 1: Refactoring/Non-Refactoring Code Samples

Method Refactoring Type	Number of Samples
Extract And Move Method	8637
Extract Method	29946
Inline Method	7098
Move Method	23835
Pull Up Method	14062
Push Down Method	8039
Rename Method	36929
Extract And Move Method	8637
Change Return Type	58308
Move And Inline Method	3751
Move And Rename Method	4826
Change Parameter Type	66803
Split Parameter	390
Merge Parameter	1213
Non Refactored Code	66803

Most of the data collected was very unbalanced with far more non-refactoring instances than refactoring instances (see Table 1) . To ensure the models are trained equally on both, the non-refactored instances were randomly downsampled to match the number of refactored instances before being used for training.

2 Models

Since this project focuses on Method-level refactorings, 13 models were created for each algorithm in order to predict the following method-level refactoring techniques Extract and move, Extract, Inline, Move, Pull up, Push down, Rename, Change Return Type, Move and Inline, Move and Rename, Change Parameter Type, Split Parameter and Merge Parameter.

Creating each model followed roughly the same process:

1. Query the Extract Method and non-Extract Method instances with their associated metrics.
2. Apply the correct label.
3. Split the data into training/validation sets with (test size: 20%).
4. Train the model on all parameter combination
5. Keep the combination with the best performance.
6. Run this model on the validation set.
7. Compare the training and testing models and calculate error.

2.1 Decision Trees

The decision tree worked very well, with a mean accuracy of 91% for every refactoring type. It also didn't take very long to train with a total training time of approximately 200 seconds. The parameters with the best results were maximum depth = 16, maximum features = log2 and entropy as the splitting criterion.

Table 2: Decision Tree Classifier Results

Name	Training Time	Validation Time	Accuracy	Precision	Recall
Extract And Move Method	4.702	0.004	0.88104	0.87386	0.8905
Extract Method	20.1134	0.006	0.89990	0.89288	0.91131
Inline Method	3.66230	0.0024	0.90669	0.89896	0.91607
Move Method	15.3581	0.0085	0.94042	0.93792	0.94491
Pull Up Method	7.7137	0.0042	0.93155	0.92045	0.94151
Push Down Method	5.1665	0.0036	0.939054	0.93950	0.939506
Rename Method	27.9543	0.01262	0.92221	0.91806	0.92595
Extract And Move Method	6.06081	0.0034	0.88509	0.88174	0.88940
Change Return Type	49.3134	0.01508	0.9277	0.913880	0.942200
Move And Inline Method	2.4823	0.0031	0.89473	0.86832	0.93075
Move And Rename Method	2.9772	0.00256	0.86897	0.85108	0.8943
Change Parameter Type	50.4801	0.01493	0.93477	0.93531	0.9339827
Split Parameter	0.72771	0.006808	0.8653	0.860759	0.87179
Merge Parameter	1.14205	0.00182	0.950617	0.94190	0.957

2.2 Random Forest

The random forest model was the most accurate out of all of them with a mean accuracy of 94.6% for each refactoring type. The model worked best with a maximum depth of 16 for all types and all except Move, Merge Parameter and Change Parameter produced better results using entropy instead of Gini as the splitting criterion. Unfortunately, since these results were run on a different computer than the others, the training and testing times are not comparable and are not shown here.

Table 3: Random Forest Results

Name	Accuracy	Recall	Precision
Extract and Move	0.919	0.937	0.903
Extract	0.933	0.956	0.916
Inline	0.935	0.968	0.908
Move	0.958	0.977	0.942
Pull Up	0.968	0.979	0.956
Push Down	0.972	0.981	0.965
Rename	0.966	0.976	0.956
Change Return Type	0.965	0.973	0.956
Move and Inline	0.927	0.936	0.920
Move and Rename	0.917	0.928	0.907
Change Parameter Type	0.962	0.986	0.941
Split Parameter	0.936	0.962	0.915
Merge Parameter	0.942	0.975	0.913

2.3 Naive-Bayes

The Gaussian Naive-Bayes classifier had the worst mean accuracy out of all of them at 64.7%, but the best recorded time coming in under 30 seconds.

Table 4: Naive-Bayes Results

Name	Training Time	Validation Time	Accuracy	Precision	Recall
Extract Method	2.2233169	0.0149	0.64429	0.59733	0.909165
Inline Method	0.54756	0.00468	0.6052	0.56332	0.931
Move Method	1.5021	0.0135	0.6261	0.5839	0.91116
Pull Up Method	0.97118	0.009	0.6753	0.61247	0.91681
Push Down Method	0.6034	0.0062	0.6731	0.6201	0.9061
Rename Method	2.83061	0.01644	0.647508	0.59368	0.918587
Extract And Move Method	0.63790	0.6503	0.5964	0.9293	
Change Return Type	4.1083	0.0274	0.64791	0.5925	0.91330
Move And Inline Method	0.4110	0.0044	0.6788	0.6201	0.9241
Move And Rename Method	0.41631	0.00436	0.65302	0.5990	0.924
Change Parameter Type	4.15922	0.03321	0.6586	0.6028	0.92488
Split Parameter	0.149188	0.0084	0.6602	0.6	0.9615
Merge Parameter	0.246	0.0033	0.6028	0.5558	0.924

2.4 Logistic Regression

The logistic regression algorithm didn't do as well as some of the others with a mean accuracy of 70.9%. It also took a very long time to train (almost 50 minutes). The best parameters varied greatly with the maximum iterations anywhere between 100 and 1000 and the c-values between 0 and 5.

Table 5: Logistic Regression

Name	Accuracy	Recall	Precision	Training Time	Validation Time
Extract and Move	0.754	0.742	0.778	52.104	0.00286
Extract	0.785	0.759	0.842	222.261	0.00617
Inline	0.670	0.672	0.664	43.352	0.0031
Move	0.699	0.728	0.646	196.119	0.00535
Pull Up	0.725	0.738	0.679	109.315	0.0042
Push Down	0.737	0.767	0.686	60.657	0.00321
Rename	0.681	0.720	0.584	405.303	0.00851
Change Return Type	0.746	0.746	0.734	590.475	0.0179
Move and Inline	0.689	0.693	0.678	22.867	0.00436
Move and Rename	0.646	0.715	0.486	42.686	0.0042
Change Parameter Type	0.782	0.777	0.788	704.16	0.0148
Split Parameter	0.686	0.746	0.564	2.072	0.00319
Merge Parameter	0.615	0.633	0.502	6.374	0.00323

2.5 Support Vector Machine

The support vector machine was also run on a different computer so training and validation times are not available. However, it had a fairly good mean accuracy- 72.5%. The best c-parameters varied between 1 and 0.1, and the maximum iterations between 2,000 and 10,000. Even with 10,000 iterations the algorithm failed to converge on some of the methods suggesting it was not the ideal model.

Table 6: Support Vector Machine

Name	Accuracy	Recall	Precision
Extract and Move	0.654	0.741	0.631
Extract	0.701	0.960	0.635
Inline	0.692	0.481	0.831
Move	0.726	0.733	0.728
Pull Up	0.763	0.774	0.750
Push Down	0.764	0.608	0.888
Rename	0.716	0.788	0.687
Change Return Type	0.687	0.765	0.656
Move and Inline	0.752	0.900	0.695
Move and Rename	0.626	0.336	0.798
Change Parameter Type	0.760	0.964	0.684
Split Parameter	0.801	0.871	0.764
Merge Parameter	0.792	0.632	0.915

2.6 Neural Network

The mean accuracy of the neural network was very good at 84.9%. It was quite slow however and took over 10 minutes to train.

Table 7: Neural Network Results

Name	Training Time	Validation Time	Accuracy	Precision	Recall
Extract And Move Method	88.3592	0.185472	0.940	N/A	N/A
Extract Method	68.36812	0.16519	0.8984	0.8931	0.90718
Inline Method	16.5713	0.085146	0.8583	0.86343	0.8159
Move Method	54.886584	0.08514	0.8953	0.86343	0.81593
Pull Up Method	34.744	0.1011	0.9295	0.93286	0.8732
Push Down Method	18.8245	0.1011	0.9453	0.922	0.9413
Rename Method	83.1514	0.9956	0.8764	0.8557	0.889131
Extract And Move Method	20.1749	0.0806	0.8863	0.9163	0.7104
Change Return Type	130.447	0.2681	0.9106	0.94031	0.81876
Move And Inline Method	9.6535151	0.06865	0.8579	0.8864353	0.74833
Move And Rename Method	11.7002	0.0721	0.8596	0.84832	0.8404
Change Parameter Type	154.620	0.9233	0.291429	0.9133	0.9377
Split Parameter	1.73847	0.0472	0.8327	0.91666	0.56410
Merge Parameter	3.3067	0.04994	0.9148	0.86178	0.8945

3 Feature Importance

Features (i.e., a numeric representation of a measurable property that is used to represent a ML problem to the model) play a pivotal role in the quality of the obtained models. In Table 8, we explore which features are considered the most relevant by the models. Such knowledge is essential because, in practice, models should be as simple as and require as little data as possible. We extracted the top features for refactoring "Extract Method" using sklearn's Permutation Feature importance..

Table 8: Top Feature Extracted (Using Permutation Feature Importance)

Feature Name	Importance
methodLoc	0.139
qtyOfCommits	0.122
startLine	0.104
authorOwnership	0.093
refactoringsInvolved	0.090
classUniqueWordsQty	0.059
methodRfc0.056	0.001
classNumberOfPublicMethods	0.052
classCbo	0.030
classLcom	0.025

4 Conclusions

All of the models did a fairly effective job at predicting refactoring instances, however the random forest algorithm was the most accurate, followed closely by the decision tree. While the Naive-Bayes had the lowest accuracy, it was also the fastest which offers its own benefits. The logistic regression model was the slowest by far, and offered a fairly middling accuracy, so it could be considered the least effective practically speaking.

Table 9: Mean Accuracy of Each Model

Model	Mean Accuracy
Decision Tree	0.911
Random Forest	0.911
Naive-Bayes	0.647
Logistic Regression	0.709
Support Vector Machine	0.725
Neural Network	0.849

The results in our project closely mirrored but did not exactly match those in the original paper. Like the original authors, we found that the Random Forest model had the highest accuracy. We also noted that Support Vector machines had high recall values but low precision. However, unlike the original results, the SVM recall was lower than that of the Random Forests. This could be due to several factors, such as the smaller dataset we used or variations in our implementation of the models.

References

[1] Fowler, Martin. "Refactoring." Refactoring.com. https://refactoring.com/

[2] Lawrence, Cate. "The Ultimate Engineer's Guide to Code Refactoring." Stepsize. https://www.stepsize.com/blog/the-ultimate-engineers-guide-to-refactoring

[3] Alshayeb, Mohammad. "Empirical investigation of refactoring effect on software quality." Science Direct. https://www.sciencedirect.com/science/article/abs/pii/S095058490900038X

[4] Aniche et al. "The Effectiveness of Supervised Machine Learning Algorithms in Predicting Software Refactoring." https://arxiv.org/pdf/2001.03338.pdf