Rochester Institute of Technology

RIT Scholar Works

Theses

5-2020

Identifying Performance Regression From The Commit Phase Utilizing Machine Learning Techniques

Max Mendelson
mwm3398@rit.edu

Follow this and additional works at: https://scholarworks.rit.edu/theses

Recommended Citation

Mendelson, Max, "Identifying Performance Regression From The Commit Phase Utilizing Machine Learning Techniques" (2020). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.

Identifying Performance Regression From The Commit Phase Utilizing Machine Learning Techniques

by

Max Mendelson

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Software Engineering

Supervised by
Dr. Mohamed Wiem Mkaouer

Department of Software Engineering
B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York

May 2020

The thesis "Identifying Performance Regression Introducing Code Changes Utilizing Machine Learning" by Max Mendelson has been examined and approved by the following Examination Committee:

Dr. Mohamed Wiem Mkaouer
Assistant Professor, RIT
Thesis Committee Chair

Dr. Yasmine El-Glaly
Professor, RIT

Dr. Ali Ben Mrad
Assistant Professor, University of Monastir

Dr. J. Scott Hawker
Associate Professor
Graduate Program Director, RIT

Acknowledgments

There are many people who have had a huge impact in my life and my college career that I would like to thank. I would first like to thank my advisor, Dr. Mohamed Wiem Mkaouer for being incredibly supportive and helpful during this entire experience. You have made this thesis work very exciting and rewarding. I would next like to thank the Software Engineering department for providing excellent resources and an incredible program for both my Undergrad and Graduate education.

I would next like to thank my parents, Cathy and Steve, for their incredible support during college and throughout my entire life. I would also like to thank my grandparents, Elaine, Walter, Judy, and Marvin for having such an integral part of my life and guiding not only myself, but my parents, to reach our full potential. Not to mention my Great-Uncle Marvin for always sparking my interest to learn and question. Without my family, including all of those not mentioned, this would not have been possible, I thank and love you all.

Abstract

Identifying Performance Regression From The Commit Phase Utilizing Machine Learning Techniques

Max Mendelson

Supervising Professor: Dr. Mohamed Wiem Mkaouer

Over the lifespan of a software application, it is inevitable that changes to the source code will be made that causes unintended slowdowns in functionality. These slowdowns are referred to as performance regression. Typically projects who are particularly concerned about performance have performance testing that are run to identify if a performance regression is introduced into the code. This is difficult however due to how time consuming and resource intensive running performance tests are when trying to simulate a realistic scenario.

The study of De Oliveira et. al. (Perphecy) [4] suggests a technique to predict the likelihood that a commit will introduce performance regression. This is done through the use of static and dynamic metrics across several projects. The study of Alshoaibi et. al. (PRICE) [2] replicates and expands the work but focuses on a single project, that being "Git". PRICE generated a large dataset consisting of benchmarks for each commit introduced into the code compared to the commit that was its predecessor.

This thesis seeks to further expand PRICE and use their collected data to create a machine learning model that can accurately predict performance regression within commits.

We have been able to successfully create these models and were able to outperform previous models at identifying performance regression introducing commits.

Contents

Acknowledgments . . . . **iii**

Abstract . . . . **iv**

1 Introduction **1**

1.1 Motivation 2

1.2 Contributions 3

1.3 Research Questions 3

2 Related Work **4**

2.1 Perphecy 4

2.2 PRICE 5

3 Methodology **7**

3.1 Approach Overview 7

3.2 Data Collection 7

3.3 Resampling Techniques 8

3.3.1 Random Undersampling 9

3.3.2 Random Oversampling 9

3.3.3 Synthetic Minority Oversampling Technique 9

3.4 Types of Experiments 10

3.4.1 Cross Validation During Resampling 10

3.4.2 Chronological 10

3.4.3 Cross Validation After Resampling 12

3.4.4 Cross Validation After Resampling Chronological 13

3.4.5 Rebalance Independently 14

3.5 Solution Evaluation 14

4 Experimental Results and Evaluation **17**

4.1 Results 17

4.2 RQ1: Which sampling technique provides the best representation for data classes? 20
4.3 RQ2: How do the classifiers perform compared with the other prediction techniques? 20
4.4 Discussion and Importance of the true positive rate 21

5 Threats to Validity **22**

6 Conclusion and Future Work **23**

Bibliography **25**

List of Tables

2.1 Metrics Descriptions and Rationales. 5

4.1 Chronological Order Results 17
4.2 CV during Resampling Results 18
4.3 CV after Rebalancing Results 18
4.4 CV after Rebalancing Chronological Results 18
4.5 Rebalance Independently Results 19

List of Figures

3.1 Cross Validation Diagram 11
3.2 Hits Per Fold Distribution 11
3.3 Chronological Experiment Diagram 12
3.4 Chronological Hits Per Test 13
3.5 Rebalance Independently Flow 15

4.1 Comparison of Best Results for Each Experiment. 19
4.2 Comparison of Paper Results 20

Chapter 1

Introduction

Software performance is an essential factor in many software projects. Typically projects that require good performance have performance tests suitable to determine if a performance regression has occurred. Performance regression takes place when software taking in the same input performing the same task becomes slower after a code change. As software evolves and matures, it is almost inevitable that some form of performance regression should occur, thus it is important to test for it. One problem, however, is that performance testing is very time consuming and very resource intensive to run. Performance tests often have to replicate real world scenarios, which turns into a large workload and not practical for developers to run as often as they should. Ideally, performance tests should be run at each commit, however, this is impractical because of how burdensome performing these tests are to run. Thus an alternative method of detecting performance regression must be created, this is what initiated the idea of Perphecy [4].

Perphecy is a system that predicts whether an incoming commit will introduce a performance regression by analyzing static and dynamic metrics from a commit and identify indicators that would signal a performance regression. PRICE [2] is a paper that took the concept of Perphecy and expanded it over a larger sample size of a specific project, that being Git. Perphecy took all of the commits from Git, up until that point, and created a program to analyze these commits and build the same indicators as from Perphecy, and attempt to predict performance regression introducing commits. In this thesis, we uses PRICE's large Git dataset and expand their prediction methods by implementing machine learning. Machine learning gives the added dimension of getting to customize the bounds

for the indicators to be project specific, thus theoretically giving a more accurate opportunity to create a predictor of performance regression.

One of the main problems of the dataset from PRICE is that it is severely imbalanced. Commits that introduce performance regression make up less than 7% of the overall dataset, this makes it very challenging to build an accurate machine learning model. For this reason, much of our research in this thesis is utilizing different rebalancing techniques and machine learning techniques to create the best predictor that we could. In this thesis, we expand what PRICE had done, make improvements to the dataset, present several methods to tackle the imbalanced data problem, and supply numerous machine learning solutions to create the best predictor.

1.1 Motivation

Software Performance is a critical aspect to the quality of software. If not detected, a performance regression could have significant impact on the effectiveness and longevity of a piece of software. Ideally, one would test for performance regression consistently, similar to executing tests on every software commit by utilizing continuous integration, this is not feasible with performance regression, however, due to how costly and time consuming it is to run performance tests. Thus, if there was a more lightweight approach that would be able to predict performance regression that could be easily run after each code commit, this would have tremendous impact. The Perphecy study is an attempt to do this exact thing and makes significant leaps to create this system. If a system was able to consistently predict performance regression, then full performance tests would only need to be run on commits where performance regression exists. This would save the company a lot of time, money, and improve software quality as they would avoid the expensive nature of unnecessary performance benchmark testing and decrease the overall commits introducing performance regression.

For the definition of performance regression in the context of this paper, we use the same definition as PRICE where it is defined as ”anytime a test's execution time is longer

than for the previous commit's in a statistically significant way." [2]

In this thesis, we focus specifically on utilizing various machine learning techniques and experiments to try to use the data gathered in PRICE and see if this will give us a more accurate predictor of performance regression. Theoretically, utilizing machine learning should create a model that gives more accurate predictions due to the more customized approach machine learning would have at identifying indicators. One main issue with utilizing machine learning for this problem is class imbalance. Due to the nature of how infrequent performance regression is, the commits that introduce performance regression in the dataset from PRICE only makes up less than 7% of the overall dataset. This makes it challenging for the classifier to identify distinguishing characteristics from these indicators and leads to a low detection rate of performance regression. Thus in this research, we focused a lot of energy in tackling the class imbalance problem in our approaches to create a successful machine learning model.

1.2 Contributions

Our main contribution is to further extend the PRICE and Perphecy papers. Our main contributions are:

1. Utilization of machine learning for performance regression introducing code changes prediction.

2. Testing multiple class rebalancing techniques along with a comparative study between various machine learning models.

1.3 Research Questions

- RQ1: Which sampling technique provides the best representation for data classes?

- RQ2: How do these classifiers perform compared with the previous prediction techniques?

Chapter 2

Related Work

In this chapter, we discuss the key related works to this thesis that our research stemmed from.

2.1 Perphecy

Perphecy is the origin of where our research extends from. Perphecy is defined as a system that ”predicts if a code commit will affect the performance of a benchmark in a benchmark suite” [4]. This regards the current latest commit and the commit that is newly being introduced into the project. Perphecy predicts whether a new commit coming in will likely break a performance test (which is what is referred by ”benchmark”), leading to a performance regression. Static and dynamic analysis is utilized to obtain these metrics and Perphecy was found to save 83% of benchmarking time while detecting 85% of performance affecting changes. Perphecy uses 8 metrics which are referred to as ”indicators” and these are what is used for determining predictions based on the old commit, new commit, and the current performance test (benchmark). In the paper titled PRICE [2], which we will discuss later within this chapter, they use 7 of the 8 benchmarks used in Perphecy for their extended research on this topic. In our research, we use PRICE's extracted data, thus we use 7 of the 8 metrics mentioned from Perphecy. The list of benchmarks that ourselves and PRICE actually used in our research and their rationales can be viewed in Table 2.1.

Table 2.1: Metrics Descriptions and Rationales.

#	Description	Rationale
1	Number of deleted functions	Deleted functions indicate refactoring, which may lead to performance changes
2	Number of new functions	Added functions indicate new functionality, which may lead to performance changes
3	Number of deleted Functions reached by the benchmark	Deleting a function which was part of the benchmark execution could lead to a performance change
4	The percent overhead of the top most called function that was changed	Altering a function that takes up a large portion of the processing time of a benchmark has a high risk of causing a performance regression because it is such a large portion of the test
5	The percent overhead of the top most called function that was changed by more than 10% of its static instruction length	Similar to metric 4, however this takes into account that the change affects a reasonable portion of the function in question. Bigger changes may mean higher risk.
6	The highest percent static function length change	Large changes to functions are more likely to cause regressions than small ones
7	The highest percent static function length change that is called by the benchmark	The same as for metric 6, but here we guarantee that the functions are actually called by the benchmark in question.

2.2 PRICE

PRICE [2] is the main paper that our research builds off of. PRICE is an extension of Perphecy and goes further in depth into the project ”Git”. PRICE replicates Perphecy's experiment but focuses only on the Git project. All of the commits from Git are obtained, up until the point PRICE did the research, and ran tests analyzing all of the commits and

collecting all of the benchmarks on these commits. The benchmarks refer to the information in Table 2.1, which include the 7 benchmarks actually used in the paper. PRICE uses 7 of the 8 metrics and omits 6 from Perphecy (which cannot be viewed in the aforementioned figure) in its actual research. PRICE was able to successfully replicates Perphecy and extracts all of the benchmark and commit comparison data. The goal of our research is to take this generated data and utilize machine learning to create a classifier that is able to accurately predict performance regression based on these benchmarks.

Chapter 3

Methodology

In this chapter, we give an overview of our approach and the various methods and experiments utilized in trying to build our machine learning models.

3.1 Approach Overview

The general idea of our approach is to try to identify performance regression via static and dynamic metrics obtained from code commits. We use the same 7 metrics as used in PRICE [2] which were selected from Perphecy [4]. The table of metrics and rationales behind them can be seen in Table 2.1. More can be learned about the selection process within the Perphecy and PRICE papers.

We utilized Microsoft Azure Machine Learning Studio to create our machine learning models. Microsoft Azure Machine Learning Studio had all of the machine learning techniques that we required pre-built so that we did not have to create these ourselves. This saved us a lot of time and allowed for quicker and more diverse experimentation as well as easy collaboration.

3.2 Data Collection

The data collection process was done within the PRICE paper [2]. PRICE replicated the Perphecy [4] data collection process to gather the dataset for this problem. The data that needed to be collected included a commit, the commit directly before that commit, and the change in the benchmarks that we can use to determine performance regression.

Overall 8798 commits were gathered. 202 of the commits did not have any tests that could be evaluated, so they were omitted. This left the total number of commits to 8596.

Running all of the performance tests were very time consuming, thus many machines were utilized to run them simultaneously over a long period of time. More on the specifics of this paralleled computing and data collecting could be found in PRICE [2].

One modification that we made to the dataset that PRICE generated was to remove invalid 0s. Each benchmark that was obtained separately gave a result of whether it introduced a performance regression (marked by a 1), or did not introduce a performance regression (marked by a 0). Thus there were numerous benchmarks for each commit, each representing a row in the dataset. There was an inconsistency, however, that if one commit had a benchmark that obtained a 1, the rest of the benchmarks for that commit could potentially be 0s. This was confusing the model because once a 1 is identified within a commit, that commit should be identified as a performance regression introducing commit, however this was not the case. For that reason, we manually removed all benchmarks that obtained a 0 where there contained at least a single 1 for that commit pair. If there were multiple 1s for different benchmarks but for the same commit pair, we would keep all instances of those 1s. This left the dataset that we used with our model with a total number of 6353 commits, omitting 2243 rows from the dataset.

3.3 Resampling Techniques

There were three different resampling techniques that we utilized to try to solve our imbalanced data problem. Those included random undersampling, random oversampling, and SMOTE. Along with these different resampling techniques, we used 5 different machine learning algorithms to try to build the best model. The different machine learning algorithms we used in conjunction with these resampling techniques, provided by Microsoft Azure Machine Learning Studio were decision forest, boosted decision tree, support vector machine, neural network, and bayes point machine.

3.3.1 Random Undersampling

RUS (Random Undersampling) [6] is a rebalancing technique that tries to solve an imbalanced data problem by providing less of the overrepresented class to reduce the imbalanced distribution with the underrepresented class. We utilized this by decreasing the number of commits that did not have performance regression issues to match about equally to the number of commits that did have performance regression issues. This would bring the dataset to about 50% commits with performance regression issues, and 50% commits without performance regression issues. The overrepresented class data points were omitted at random.

3.3.2 Random Oversampling

ROS (Random Oversampling) [6] is a rebalancing technique that tries to solve an imbalanced data problem by providing more of the underrepresented class to reduce the imbalanced distribution with the underrepresented class. It achieves this by duplicating the underrepresented class, in this case commits that introduced performance regression, to more closely match the number of commits that did not introduce performance regression. This would bring the dataset to about 50% commits with performance regression issues, and 50% commits without performance regression issues.

3.3.3 Synthetic Minority Oversampling Technique

SMOTE (Synthetic Minority Oversampling Technique) [3] is a rebalancing technique that tries to solve an imbalanced data problem by providing more of the underrepresented class via generated instances that are similar to existing minority class instances. The algorithm we use with SMOTE to generate instances is K-Nearest Neighbors Algorithm (KNN). With SMOTE you can modify how far the ”nearest neighbors” should be, which represents how much of a variation the generated data should be from existing data. We used SMOTE to get the number of the minority class to be about the same as the majority class. This would bring the dataset to about 50% commits with performance regression issues, and

50% commits without performance regression issues.

3.4 Types of Experiments

During Experimentation, we tried a variety of methods to see how the models and results would be impacted in different scenarios. This section will cover all of the various experiments that we have utilized with our dataset.

3.4.1 Cross Validation During Resampling

Cross validation during resampling [5] [7] [1] [8] is an approach that separates the data into about equally represented folds and uses select folds as test data and the remaining folds as training data. In our experiments, we did 10-fold cross validation. In 10-fold cross validation we use 9 folds for training and 1 fold for testing. We run this experiment for each possible testing fold, doing 10 tests overall, and average out all of the individual results to get our final average result numbers. A diagram displaying the iteration cycle of CV during resampling can be seen in Figure 3.1. For more details on CV during resampling, see Santos et. al. [9]. We use cross validation with all 3 of the Rebalancing techniques as well as all 5 of the Machine Learning methods mentioned prior. The distribution of the number of 1s (representing a commit introducing a performance regression) per fold can be seen in figure 3.2.

3.4.2 Chronological

In order to test if the chronological order of when training data was utilized by the model had an impact on how effective the model would be, we implemented chronological testing. Chronological order refers to the dates that the commits were actually submitted to Git. Chronological testing is a method that we use that is similar to cross validation in that we do multiple testings over a 10-fold split, however we make sure that the order of the folds are not shuffled. The way we achieve this is by splitting the commits into 10 equal folds in chronological order, for example, all commits in fold 3 were committed before

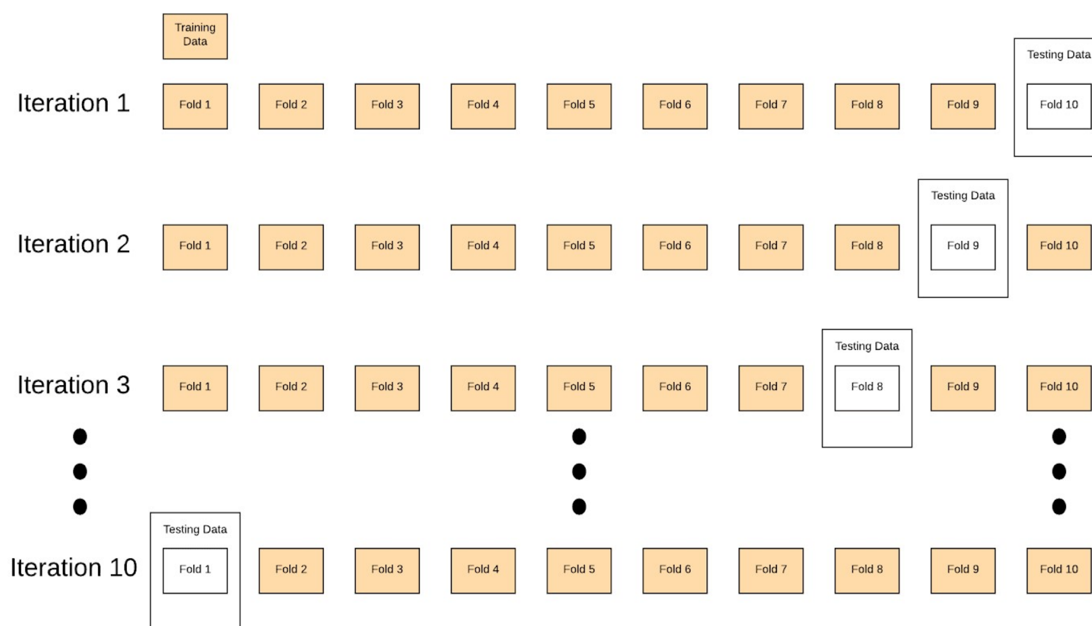

Figure 3.1: Cross Validation Diagram

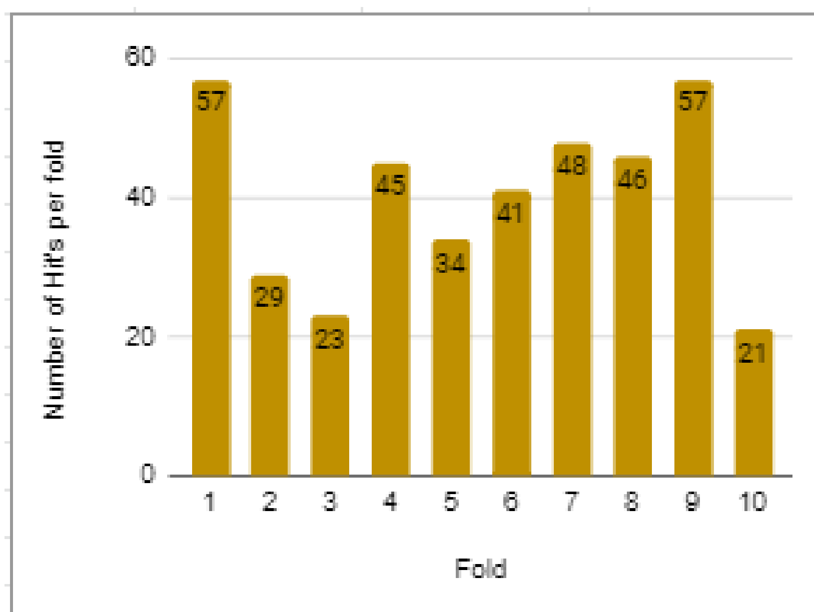

Figure 3.2: Hits Per Fold Distribution

commits in fold 4 and so on. There are 5 overall tests, the first test consists of folds 1-9 as the training data and fold 10 as the testing data. The second tests consists of folds 1-8 as the training data and 9-10 as the testing data. This is done until folds 1-5 are used as the

training data and fold 6-10 are used as the testing data. A diagram displaying the iteration cycle of chronological testing can be seen in Figure 3.3. We use chronological testing with boosted decision tree, decision forest, and SVM as well as all 3 rebalancing techniques. The distribution of the number of 1s (representing a commit introducing a performance regression) per fold can be seen in figure 3.2. The distribution of how many 1s were in each test set could be seen in figure 3.4.

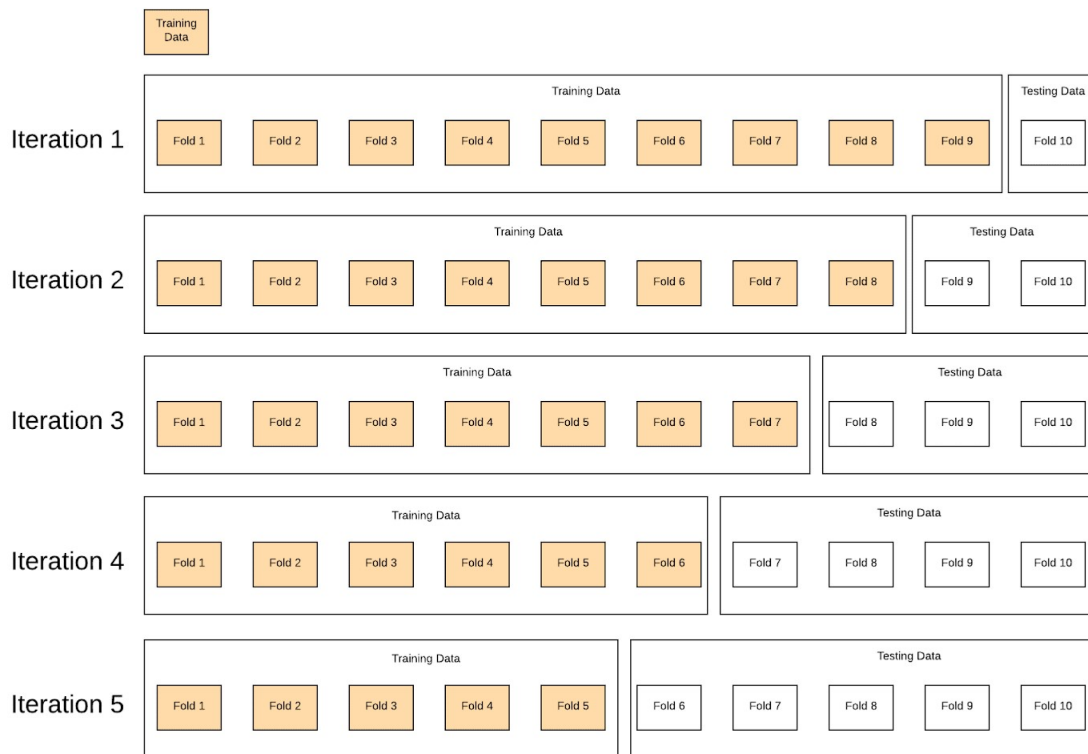

Figure 3.3: Chronological Experiment Diagram

3.4.3 Cross Validation After Resampling

Cross validation (CV) after resampling is an approach that does the rebalancing technique before the training and testing data are split up. After the rebalancing technique is applied, whether it be RUS, ROS, or SMOTE, the testing data and training data will both be equally balanced when the model is being generated and assessed. For more details on CV after resampling, see Santos et. al. [9]. We use CV after resampling with all 5 training algorithms

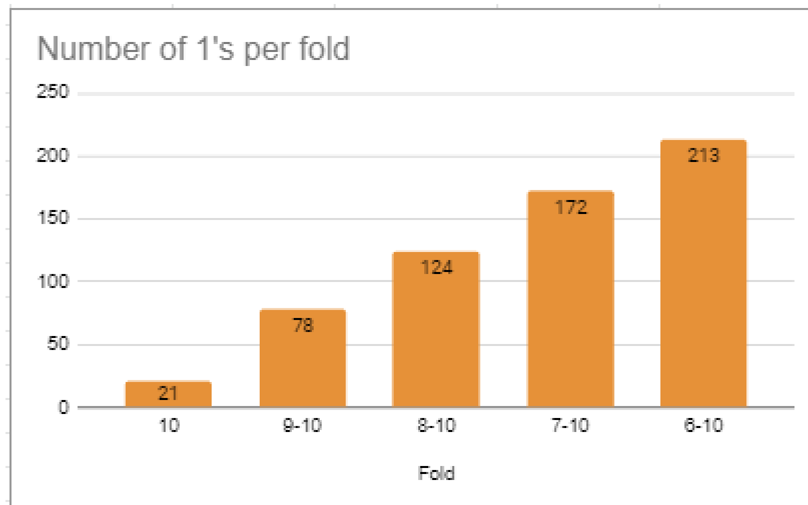

Figure 3.4: Chronological Hits Per Test

as well as all 3 rebalancing techniques. The distribution of the number of 1s (representing a commit introducing a performance regression), before the rebalancing technique was applied, per fold can be seen in figure 3.2.

3.4.4 Cross Validation After Resampling Chronological

Cross Validation after resampling chronological combines the chronological method as well as the CV after resampling method. This method applies the rebalancing technique, which is only Undersampling for this method, at the start before the testing and training data are split up. After this step, we do a similar cross validation to chronological and keep the dataset in order when doing our experiments. First we use fold 1-9 as the training data and fold 10 as the testing data, then we use fold 1-8 as the training data and 9-10 as the testing data, and so on until 1-5 are the training folds and 6-10 are the testing folds. A diagram displaying the iteration cycle of CV after resampling chronological can be seen in Figure 3.3. For more details on CV after resampling, see Santos et. al. [9]. This method can only be done with Undersampling because we cannot keep datasets that use SMOTE or Oversampling in order as SMOTE uses generated data that does not have a chronological order, and Oversampling would lead to biased test results. We only use this method with

decision forest to compare it with traditional CV after resampling. The distribution of the number of 1s (representing a commit introducing a performance regression), before the rebalancing technique was applied, per fold can be seen in figure 3.2.

3.4.5 Rebalance Independently

Rebalance Independently is a technique that splits the training and testing data before any rebalancing technique is performed. After the data is split into training and testing, we apply the same rebalancing technique to both the training and testing dataset independently of one another. We utilize 10-fold cross validation with this method, thus performing 10 tests overall. For example, one iteration of this could be to split fold #1 and identify it as our testing data, and identify folds #2-#10 as our training folds. We apply SMOTE to the testing dataset containing fold #1, and then we apply SMOTE to the training dataset consisting of folds #2-#10. The model is then created and then its performance is calculated via the independently rebalanced testing dataset that comprised of fold #1. A diagram displaying the iteration cycle of rebalance independently can be seen in Figure 3.1. A diagram displaying the flow of rebalance independently can be seen in figure 3.5. We use Rebalance Independently with decision forest and all 3 rebalancing techniques. The distribution of the number of 1s (representing a commit introducing a performance regression) per fold can be seen in figure 3.2.

3.5 Solution Evaluation

The main metrics for evaluation and comparison within this thesis are true hit rate and true dismiss rate.

True hit rate, also known as recall, indicates the number of correctly detected commits to the total number of commits encountering performance regression [2]. The true hit rate is indicated by a number in-between 0.0 and 1.0. Having a hit rate of 0.0 means that no problematic commits were correctly identified as introducing performance regression. Having a hit rate of 1.0 means that all problematic commits were correctly identified as

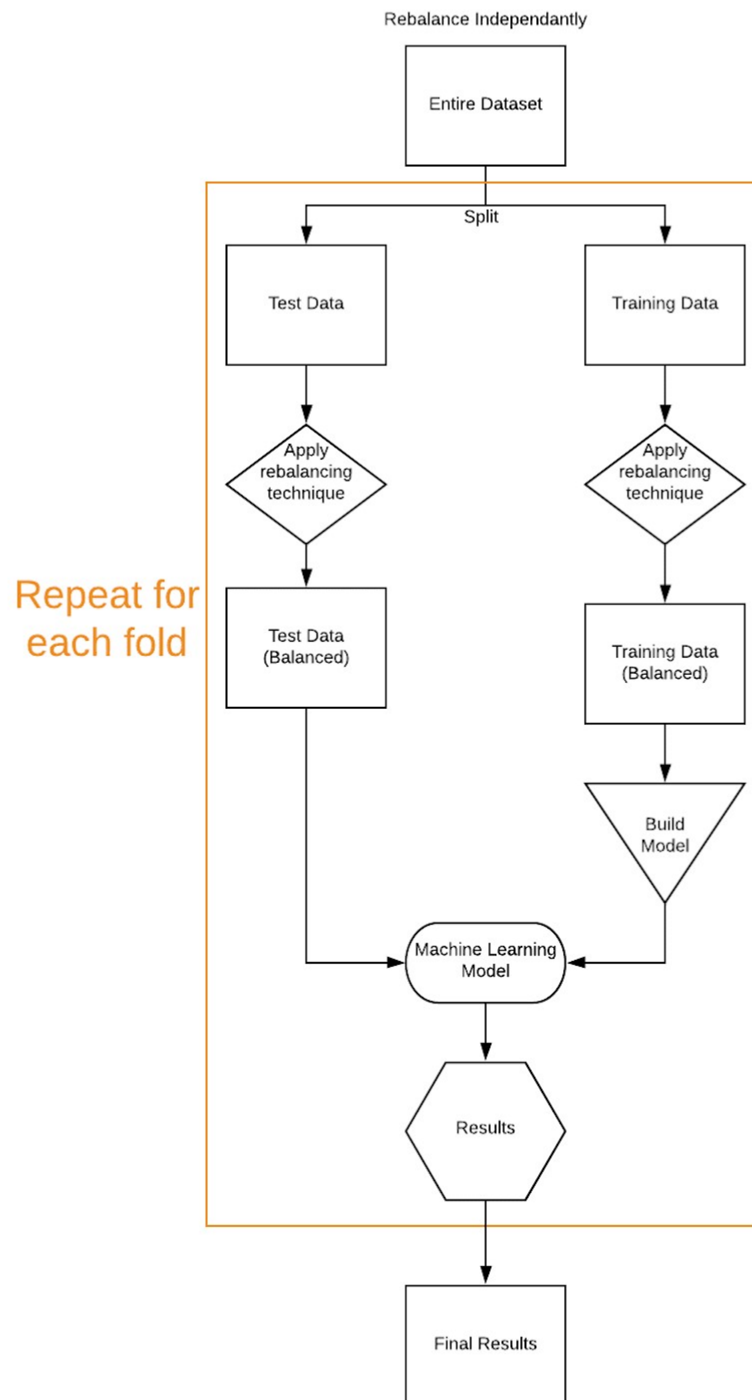

Figure 3.5: Rebalance Independently Flow

introducing performance regression.

True dismiss rate is the number of commits classified not to be introducing performance

regression to the total actual number of stable, not-performance regression introducing commits [2]. The dismiss rate is indicated by a number in-between 0.0 and 1.0. Having a dismiss rate of 0.0 means that no non-problematic commits were correctly identified as not introducing performance regression. Having a dismiss rate of 1.0 means that all non-problematic commits were correctly identified as not introducing performance regression.

Ideally the hit and dismiss rates would both be 1. Due to the nature of these metrics and how they are conflicting, we are searching for an optimal solution which would consist of having a high true hit rate and high true dismiss rate.

In our research we use additional metrics as well for evaluation. We use f-score, precision, and false negative rate.

The following definitions represent the formulas we used within our calculations.

True Hit Rate (Recall):

$$TrueHitRate = \frac{\text{TP}}{\text{TP+ FN}}$$

True Dismiss Rate:

$$TrueDismissRate = \frac{\text{FP}}{\text{FP+ TN}}$$

Precision:

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

F-Score:

$$F_1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

False Negative Rate:

$$FalseNegativeRate = \frac{\text{FN}}{\text{FN + TP}}$$

Chapter 4

Experimental Results and Evaluation

In this chapter, we show the results that we obtained from our experiments as well as analysis giving context to the data.

4.1 Results

The following tables including tables 4.1-4.5 are the results from all 5 of our various methods of experiments. Under each method of experiment, the training algorithm used and rebalancing technique are listed. The background behind each type of experiment and technique can be seen in the ”Methodology” chapter of this thesis.

Experiment Number	Type of Experiment	Training Algorithm	Rebalancing Technique	True Hit Rate	True Dismiss Rate	F-Score	Precision	False Negative Rate
1	Chronological Order	Boosted Decision Tree	RUS	0.1787	0.1821	0.1781	0.0450	0.8213
			SMOTE	0.2244	0.1545	0.1763	0.0703	0.7756
			Original	0.1010	0.0378	0.0501	0.1394	0.8990
			ROS	0.1697	0.1485	0.1447	0.2657	0.8303
		Decision Forest	RUS	0.4420	0.6445	0.4355	0.0915	0.5580
			SMOTE	0.1438	0.1325	0.1359	0.0631	0.8562
			Original	0.0454	0.0145	0.0202	0.3266	0.9546
			ROS	0.0970	0.1377	0.1038	0.2388	0.9030
		SVM	RUS	0.3422	0.7088	0.2810	0.1051	0.6578
			SMOTE	0.4791	0.5839	0.4587	0.0851	0.5209
			Original	0.0000	1.0000	0.0000	0.0000	1.0000
			ROS	0.6324	0.3478	0.3884	0.0812	0.3676
	Maximum value			0.6324	1.0000	0.4587	0.3266	1.0000

Table 4.1: Chronological Order Results

The main points of data that we look at are the true hit rate and true dismiss rate so that we can have an accurate comparison to PRICE as these are the metrics used in that paper. We have also calculated and gathered other metrics as well including f-score, precision, and false negative rate. The formulas for all of these calculations can be seen in the

Experiment Number	Type of Experiment	Training Algorithm	Rebalancing Technique	True Hit Rate	True Dismiss Rate	F-Score	Precision	False Negative Rate
2	CV During Resampling	Boosted Decision Tree	RUS	0.4500	0.7013	0.4753	0.0924	0.5500
			SMOTE	0.1794	0.8612	0.2845	0.0807	0.8206
			Original	0.1419	0.9306	0.2401	0.1432	0.8581
			ROS	0.2875	0.7695	0.3711	0.1177	0.7125
		Decision Forest	RUS	0.4430	0.6900	0.5010	0.0980	0.5570
			SMOTE	0.2472	0.7682	0.3295	0.0857	0.7528
			Original	0.1040	0.9620	0.1781	0.2458	0.8960
			ROS	0.3160	0.7820	0.4140	0.2070	0.6840
		SVM	RUS	0.6080	0.5040	0.4830	0.0780	0.3920
			SMOTE	0.6359	0.5247	0.4761	0.0744	0.3641
			Original	0.0000	1.0000	0.0000	0.0000	1.0000
			ROS	0.4300	0.6370	0.3610	0.0620	0.5700
	Maximum value			0.6359	1.0000	0.5010	0.2458	1.0000

Table 4.2: CV during Resampling Results

Experiment Number	Type of Experiment	Training Algorithm	Rebalancing Technique	True Hit Rate	True Dismiss Rate	F-Score	Precision	False Negative Rate
3	CV After Resampling	Boosted Decision Tree	RUS	0.8000	0.7229	0.7665	0.7356	0.2000
			SMOTE	0.8789	0.8445	0.8601	0.8422	0.1211
			Original	0.3125	0.9832	0.4743	0.5556	0.6875
			ROS	0.9981	0.9714	0.9846	0.9679	0.0019
		Decision Forest	RUS	0.7500	0.7711	0.7547	0.7595	0.2500
			SMOTE	0.8540	0.8092	0.8307	0.8086	0.1460
			Original	0.1625	0.9756	0.2786	0.3095	0.8375
			ROS	1.0000	0.9756	0.9877	0.9725	0.0000
		SVM	RUS	0.6625	0.5542	0.6235	0.5889	0.3375
			SMOTE	0.6836	0.4798	0.5639	0.5862	0.3164
			Original	0.0000	1.0000	0.0000	0.0000	1.0000
			ROS	0.4391	0.7000	0.5397	0.5582	0.5609
		Neural Network	RUS	0.0000	1.0000	0.0000	0.0000	0.5258
			SMOTE	0.8054	0.4563	0.5825	0.5649	0.1946
			Original	0.0000	1.0000	0.0000	0.0000	1
			ROS	0.4742	0.8143	0.5994	0.6879	1.0000
		Bayes Point	RUS	0.5375	0.6000	0.5670	0.5309	0.4634858812
			SMOTE	0.5532	0.6571	0.6007	0.5858	0.4467881112
			Original	0.0000	1.0000	0.0000	0.0000	1
			ROS	0.5365	0.7017	0.6081	0.6082	0.4625
	Maximum value			1.0000	1.0000	0.9877	0.9725	1.0000

Table 4.3: CV after Rebalancing Results

Experiment Number	Type of Experiment	Training Algorithm	Rebalancing Technique	True Hit Rate	True Dismiss Rate	F-Score	Precision	False Negative Rate
4	CV After Resampling Chronological	Decision Forest	RUS	0.5790	0.5690	0.4780	0.5200	0.4210
	Maximum value			0.5790	0.5690	0.4780	0.5200	0.4210

Table 4.4: CV after Rebalancing Chronological Results

”Methodology” chapter of this thesis.

The best results overall are very clearly the CV after resampling experiment labeled experiment #3 in figure 4.3. This experiment technique, however, could lead to overoptimistic results and overfitting [9]. For this reason, we must take that into consideration with

Experiment Number	Type of Experiment	Training Algorithm	Rebalancing Technique	True Hit Rate	True Dismiss Rate	F-Score	Precision	False Negative Rate
5	Rebalance Independently	Decision Forest	RUS	0.4500	0.7220	0.5010	0.6270	0.5500
			SMOTE	0.4900	0.7730	0.5640	0.6850	0.5100
			ROS	0.1980	0.8390	0.2920	0.4450	0.8020
	Maximum value			0.4900	0.8390	0.5640	0.6850	0.8020

Table 4.5: Rebalance Independently Results

the overall analysis. Chronological order did not seem to have much of an impact on the results. If we compare experiment #1 in figure 4.1 and experiment #2 in figure 4.2, the results are very similar in most of the cases and the best result in regards to true hit rate is actually slightly higher with non-chronological testing.

A comparison of the best rebalancing / machine learning techniques, in reference to the highest true hit rate, for each experiment can be seen in figure 4.1. The best model for chronological utilized SVM and random oversampling (ROS). The best model for CV during resampling utilized SVM and SMOTE. The best model for CV after resampling utilized a boosted decision tree (BDT) and SMOTE. CV after rebalancing only had 1 method, which was decision forest (DF) and random oversampling (ROS), so this was included. The best model for rebalance independently was decision forest (DF) and SMOTE.

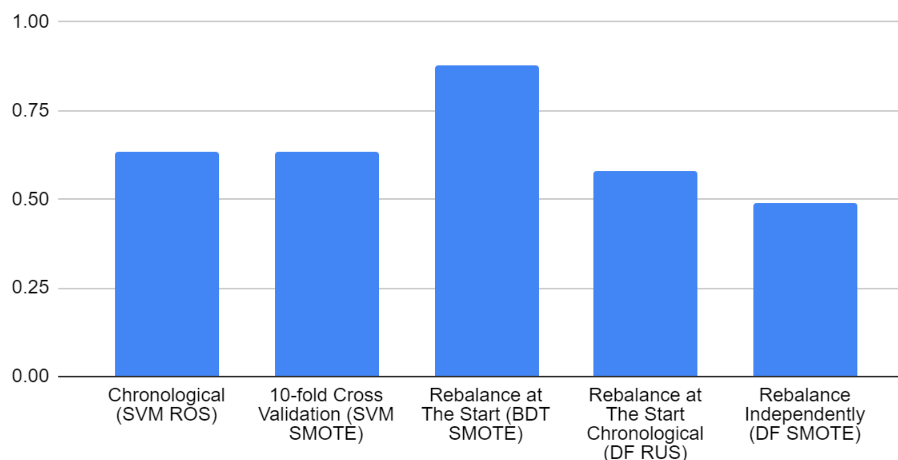

Figure 4.1: Comparison of Best Results for Each Experiment.

4.2 RQ1: Which sampling technique provides the best representation for data classes?

The experiment that resulted in the best model was by far the CV after resampling experiment. This experiment however is likely overoptimistic and has overfitting [9]. Thus when the next best experiment and techniques are assessed, standard cross validation during oversampling is the selection. The best example of CV during resampling, SVM with SMOTE, has a higher true hit rate, true dismiss rate, f-score, and precision to chronological order's best, SVM with ROS, which is the next best experiment. CV during resampling is also easier to apply since order does not matter, making it more accessible to try different experiments with.

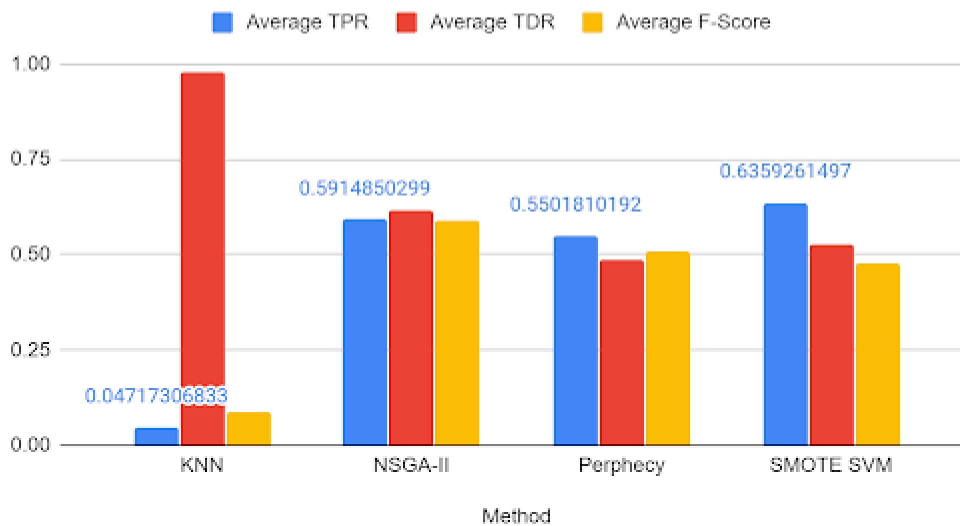

Figure 4.2: Comparison of Paper Results

4.3 RQ2: How do the classifiers perform compared with the other prediction techniques?

Our best resulting model was better than the existing prediction techniques offered by previous papers. We judge this based on true positive rate (TPR), which is the same as true

hit rate, as this is what determines whether a performance regression introducing commit is correctly classified. For more on why we identify TPR as our main identifier and reasons for looking at other identifiers as well, visit the ”Discussion” chapter of this thesis. For a comparison with the other prediction techniques from other papers with this model, view figure 4.2. We observe from this figure that our model was able to provide competitive results in comparison with existing related studies.

4.4 Discussion and Importance of the true positive rate

Evaluation in this type of problem is difficult. When choosing what metrics are most important, the context of the project is the key factor. When deciding how to balance the weight of all of our metrics, decisions have to be made on what we feel is the most important. Having a high TPR means that there will be less false negatives, meaning less performance regression introducing commits misclassified as not harmful. Having a high true dismiss rate (TDR) means that there will be less false positives, meaning less non-performance regression introducing commits being misclassified as harmful. In a really time-sensitive project where performance regression issues could have significant or even life threatening impact, it is important to have the highest TPR possible, even if it results in a lower TDR. For some projects, additional unnecessary performance testing time is worth the mitigation of misclassifying a performance regression introducing commit. If performance regression is not as devastating to a project, it may not be a bad idea to reduce the TPR to get a better tdr so that not as many unnecessary performance tests are done with the caveat of potential performance regression introducing commits are mistakenly accepted into the source code. This is where f-score comes into play because it accounts for both TPR and TDR. This is really project dependant and likely it is best to give the potential user of the software the option. Some projects would benefit more from maximizing TPR, some projects would benefit more from maximizing f-score. In regards to this research however, the objective is to identify performance regression introducing commits, thus we weight TPR as the most important metric.

Chapter 5

Threats to Validity

In this chapter, we present the main factors that could impact our results and possible application of our research.

One of our main threats to our most successful method, CV after resampling, is that rebalancing prior to separation of test data and training data could lead to overfitting and overoptimistic results [9]. This research suggests performing CV (cross validation) during oversampling, which is our CV during resampling, rather than CV after oversampling, which is represented by this experiment.

Another threat that we have is the small sample size of performance regression inducing commits and a severe imbalance with this dataset. With our example with Git, we have only 401 commits that introduce performance regression vs. the 6353 total rows. This may show differing results to either projects that have more performance regression introducing commits or more commits to analyze overall.

One threat that this research has is the scalability of utilizing machine learning with newer projects. When a project initially is developed, there won't be enough commits to accurately build a model off of. We don't currently know the threshold of what a reliable model can be built at in regards to either total number of commits or total number of performance regression introducing commits.

Chapter 6

Conclusion and Future Work

The objective of this research was to apply machine learning and solve the imbalanced data problem associated with identifying performance regression introducing commits. This thesis is an extension of the PRICE and Perphecy papers and we were able to take their work and expand it to utilize machine learning to solve this problem. We accomplished this by running 5 different types of experiments, 5 different types of training algorithms, and 3 different rebalancing techniques. We were able to find the best combination of machine learning model as well as rebalancing technique to solve this problem. Overall we were successful in using machine learning with numerous rebalancing and experiment techniques to create models that were able to identify performance regression at a higher rate than previous work.

Regarding future work, there are a lot of potential areas to further research. One clear area of research to back up the results of this thesis would be to run tests on more projects. Gathering metrics on numerous projects would show how effective our techniques would be on a greater scale and provide a lot of conclusions on the direction to take for predicting performance regression. Another area of potential future work is to obtain more commits from Git and use them as test data to see how the model interacts with newer commits. These commits can also be put into the training data as well and potentially improve our machine learning model. The last area of interest for the next step of future work would be to discover more rebalancing techniques and machine learning techniques. As new techniques for rebalancing are published, this provides alternative methods that could potentially perform better than our techniques. Due to the nature of this type of problem and the

fact that there will likely always be a class imbalance issue with performance regression, being aware of new rebalancing techniques could dramatically improve the results.

Bibliography

[1] U Rajendra Acharya, Vidya K Sudarshan, Soon Qing Rong, Zechariah Tan, Choo Min Lim, Joel EW Koh, Sujatha Nayak, and Sulatha V Bhandary. Automated detection of premature delivery using empirical mode and wavelet packet decomposition techniques with uterine electromyogram signals. *Computers in biology and medicine*, 85:33–42, 2017.

[2] Deema Alshoaibi, Kevin Hannigan, Hiten Gupta, and Mohamed Wiem Mkaouer. Price: Detection of performance regression introducing code changes using static and dynamic metrics. In *International Symposium on Search Based Software Engineering*, pages 75–88. Springer, 2019.

[3] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[4] Augusto Born De Oliveira, Sebastian Fischmeister, Amer Diwan, Matthias Hauswirth, and Peter F Sweeney. Perphecy: performance regression test selection made simple but effective. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 103–113. IEEE, 2017.

[5] Paul Fergus, Pauline Cheung, Abir Hussain, Dhiya Al-Jumeily, Chelsea Dobbins, and Shamaila Iram. Prediction of preterm deliveries from ehg signals using machine learning. *PloS one*, 8(10), 2013.

[6] Nathalie Japkowicz. The class imbalance problem: Significance and strategies. In *Proc. of the Int'l Conf. on Artificial Intelligence*. Citeseer, 2000.

[7] K Usha Rani, G Naga Ramadevi, and D Lavanya. Performance of synthetic minority oversampling technique on imbalanced breast cancer data. In *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 1623–1627. IEEE, 2016.

[8] José A Sáez, Bartosz Krawczyk, and Michał Woźniak. Analyzing the oversampling of different classes and types of examples in multi-class imbalanced datasets. *Pattern Recognition*, 57:164–178, 2016.

[9] Miriam Seoane Santos, Jastin Pompeu Soares, Pedro Henrigues Abreu, Hélder Araújo, and Joao Santos. Cross-validation for imbalanced datasets: Avoiding overoptimistic and overfitting approaches [research frontier]. *ieee ComputatioNal iNtelligeNCe magaziNe*, 13(4):59–76, 2018.