5-2020

Detecting Performance Regression Inducing Code Changes Using Static and Dynamic Metrics

Hiten Gupta
hg1928@rit.edu

Detecting Performance Regression Inducing Code Changes Using Static and Dynamic Metrics

by

Hiten Gupta

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 "Detecting Performance Regression Inducing Code Changes Using Static and Dynamic Metrics" by Hiten Gupta has been examined and approved by the following Examination Committee:

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

Dr. Abdulmutalib Masaud-Wahaishi
Assistant Professor, RIT

Dr. Ilyes Jenhani
Assistant Professor,
Prince Mohammad Bin Fahd University
Saudi Arabia

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

Dedication

To my family who supported me, my teachers who instilled an early love of learning in me, and my friends who walked with me.

Acknowledgments

There are many people who have played a role in helping and supporting me through my Masters studies. First, I'd like to thank my advisor, Dr. Mohamed Wiem Mkaouer for introducing me to the topic of Performance Regression, as well as all his encouragement, advice, and guidance throughout the work on my thesis. I'd like to thank my Graduate Program Director Prof. J. Scott Hawker for his guidance and support through out my journey at RIT.

Additionally, I'd like to thank all of the RIT faculty and staff, in Software Engineering, who have provided support during my time at RIT.

Next, I'd like to thank my father, Mr. Anil Gupta, for passing on his passion for technology to me, for the never-ending support through all the good and bad days. I'd also like to thank my mother, Sangeeta Gupta, and the rest of my family for their unwavering love and support - this work would not have been possible without them.

Finally, I would like to thank my sisters Dr. Nupur Gupta, Surbhi Gupta and brother-in-law Dr. Rajat Kapoor for supporting me throughout, for their endless and never-ending support throughout this work and my career.

Abstract

Detecting Performance Regression Inducing Code Changes Using Static and Dynamic Metrics

Hiten Gupta

Supervising Professor: Dr. Mohamed Wiem Mkaouer

Performance regression testing is a cost-intensive task as it delays the system development. The process, if performed in Iterations, significantly slows down the developer's pace. Hence, it is essential to execute the performance tests only on the new commit and not the whole system, as regression is induced by a newly made change to the system. This work presents a novel contribution to the detection of performance regression inducing code changes to solve the optimization problem. In this study, we combine the static and dynamic metrics as features to train classifiers to predict the performance regression if introduced by the newly made change.

To early predict the performance regression inducing code changes, we teach multiple classifiers and compare them with previous techniques. The classification of this type of data is difficult because of the Class Imbalance Problem. In any code base, over some time, it is ensured that the number of problematic commits is lower than the number of non-problematic commits. This creates the class imbalance problem as the number of problematic changes would be severely small as compared to the non-problematic changes. We tackle the class imbalance problem by using various resampling techniques: ROS, RUS, SMOTE, and compare them with each other and the original dataset. The project used to evaluate our approach is *Git*.

Our approach shows impact and effectiveness to save the testing time of the performance tests and also to solve the class imbalance problem to aid further studies and state-of-the-art procedures.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter, we look at the background of performance tests and bugs and their value in a project-oriented with Continuous Integration.

Software Engineering projects have a different scale. The plans that have a large scale use continuous Integration to bundle the code together. There are many ways in which the performance bugs can occur. When the execution time of the program is slow, it is categorized as a performance bug. The projects are accompanied by performance tests to test the performance of the ever-evolving software and surface the performance bugs. The software's memory usage is also a metric for its performance[35].

Performance tests are sometimes precise, making it tedious to manually look at the changes and require a lot of developer time for their execution along with the scenarios being realistic. This has been a concern, which requires attention and work, although it is expensive in resources as well as time. [7] [19]

Over some time, as the software changes and underlying code changes are introduced, the software either has slowdowns or speed ups because of the software changes. These slows downs and speed ups are a cause of the ever-changing nature of the software. This phenomenon is called performance regression. Performance regression is introduced in the application because of the code changes.

Chapter 2

Background & Related Work

2.1 Performance Testing

Performance Regressions testing is difficult essentially because of two main reasons: (1) Opposite to functional testing; performance regressions are embedded in long-running code and large input sizes. (2) The uncertainty in modern systems makes it difficult to have unreliable detection of the performance bugs. [26]. State-of-the-art approaches do work on easing the complications of the Regression Benchmarking but do not reduce the cost associated to it like BEEN[27], which is infrastructure on a general level to do benchmarking, takes care of deployment, runs benchmarks, and collects, evaluates, and visualizes results for the performance regression testing. Another infrastructure is DataMill[15], which extends these ideas for benchmarking, focusing on several environmental factors related to performance regression[33].

Some large scale systems make their framework for regression benchmarking, like, Mozilla's Talos performance regression detection system[3], runs a performance test every time a change is made to the Firefox source repository. The methodology used by Talos is having the prime focus on running performance tests, rather than benchmarking, which cuts down the cost of time. Still, this way of testing requires the developers to write performance tests for every critical aspect of the application, and this does not cover the performance regression embedded in the tests, which are run with large inputs.

Similarly, The Linux kernel performance project[1] conducts performance regression in a weekly manner and with each significant kernel release[11]. Despite the significant size of their test suite, they find that their tests only cover a portion of performance regressions, and they call for volunteers to contribute additional resources to enable more extensive benchmarking[14].

2.1.1 Regression Test Selection

Traditional regression selection techniques mainly aim at the correctness of tests, not on the performance regressions. Some examples like Ekstazi[20], look at dependencies of the tests and check the availability of the tests in terms of reachability.

One work in the area of regression test selection is PRA ('performance risk analysis')[13], which is close to Perphecy[14]. Huang *et al.*[13] presents "whiteboard approach", performance regression test selection that requires a static analysis to determine the "expensiveness" and "frequency" of the code change made in the newer commit.

To tackle the imprecision in PRA for programs that use dynamic languages to write the code, Sandoval *et al.*[37] proposes the solution of "horizontal profiling". The methodology is to profile a *prior* version of the application to determine the precise execution time for each code block.

Perphecy uses the general approach to gather dynamic information about the execution of the previous commit. They train an application-specific prediction model that uses the available information to predict which performance tests to run for a given commit.

2.2 Class Imbalance

In this work, we handle the class imbalance problem along with the comparison of different classification techniques that are widely used in the machine learning ecosystem, explained in section 4.4.1.

Imbalance in data occurs when there is a difference between the classes of the given problem[25]. In reference to binary classification, inequality in the data is said to exist when there is an under-represented class(minority) compared to the other class(majority)[24]. Prediction is difficult when using an imbalanced dataset, as the models are biased towards the majority class. There are several approaches to handle class imbalance[6][21][] which can be categorized in terms of data level approaches, where the data is pre-processed and ready to be balanced, where the classifiers are robust enough to adapt to the varying characteristics of the information [10][31][18][8].

The data level approaches are convenient, and there are cost-effective are proven to be independent of the classifiers [28][30]. The data level approaches can mainly be divided into two categories:

1. Undersampling

2. Oversampling

There is an inclination in the usage of Oversampling techniques as they can eliminate the class imbalance problem without getting rid of some of the critical majority examples[22]. In this work, we compare all the techniques that are encompassed by the data level approaches by our research questions, and Oversampling the data has proven to be better than Undersampling in representing the data classes. *This work also verifies that* - ”Cross Validation being used with oversampling techniques on the whole data set and then the training should take place” is a *misconception*[17][36][4][34].

There have been past work in addressing the class imbalance problem in the performance testing selection process. For example, Oliveira *et al.*[14] in the perphecy talk and highlight the Regression test selection process and use indicators for their study. The indicators that they use are elaborated by Luo *et al.*[29] by making a tool call PerfImpact. The indicators in Perphecy are the ones that serve as the feature set for this study and classifier. In PerfImpact, the indicators are carried out, looking at the effect of the introducing changes in the application.

Alshoaibi *et al.*[5] has conducted a similar study to study and make a classifier using the curated dataset from the Perphecy paper and the same feature set. In this approach they generate a rule for classifying a new commit. The rule generation process uses Evolutionary Algorithm NSGA-II[16] to train the classifier to make the commits which evolve the given metrics to generate a detection rule that maximizes the detection of problematic code changes, but, this approach does not look at resampling techniques for training the classifier, which is a essential part of training a binary classifier.

In our work, we extend on the handling the classification difficulty and enhance the classifier to correctly identify problematic commits by comparing its performance and also see if the traditional classifiers can be used for training the model. For classification we use the metrics to train the classifier for classifying the positive class(problematic commits) and resample the data to eliminate class imbalance in the dataset.

We study both the categories of data level approach of resampling, undersampling, and oversampling, where oversampling is carried out using ROS(replication) and SMOTE(KNN). We use this approach to see which technique represents the data class best for us; the results and methodology of the methods are elaborated in chapter 5.

Chapter 3

Research Objectives

3.1 Motivation

Testing after making changes to the software is a conventional practice in every aspect and not just regression. In the world of continuous integration, it is very much needed as the face the software keeps changing and also its nature.

The main aim of this work is to find a way to reduce the time of testing performance of software, also not lose the pace of testing along with making classifications as to what needs to be tested. The PRICE paper talks about different techniques that can be used to build a classifier that can classify and predict which change can introduce performance regression. A system that could reliably predict whether the change will cause performance regression or not, then this system would reduce the developer time in a significant amount.

A performance regression, in the context of this work, is defined as anytime a test's execution time is longer than for the previous commit's in a statistically significant way.

3.2 Contribution

Our primary objectives and contributions to this work are to solve the class imbalance problem when training a classifier to predict performance regression. We use machine learning algorithms mentioned in section 4.4.1 to train the classifier, for marking a commit as problematic commit and non-problematic commit. The machine learning techniques perform in classification by using the static and dynamic metrics as features.

1. **Prediction for Problematic changes.** In this work, we check for different ways of training the classifier with the imbalanced data using the feature-set of dynamic metrics. As the curated dataset was prepared by perphecy paper [23]. This dataset takes dynamic metrics as the features for a set of commits that are chronologically ordered. These features are used to train the classifier to predict future commits as problematic or non-problematic.

   The classified commit pair will indicate which performance test needs to be executed and this saves the developer time by using machine learning to filter non-problematic commits.

2. **Class Imbalance.** In a project, the changes that introduce performance regression are way low than the changes that are non-problematic. This data is severely imbalanced, which makes it difficult for the classifier to be trained for the prediction. The class in the data is problematic/non-problematic change, which makes arises the need for binary classification, hence, we used all the two-class classification algorithms (section 4.4.1). This work tackles ways to handle this imbalance for the classifier and the challenges faced in training the classifier.

The data imbalance can be seen in the Figure 4.1 that the number problematic changes in the dataset are scattered and this depicts the realistic approach and varying nature of the project.

3.3 Research Questions

1. **RQ1. Which sampling technique provides best representation for data classes?**

   This research question goal is to find the best sampling technique to balance PRICE data set in order to prepare the data set for the classification stage. We are interested in identifying the effectiveness of various re-balancing techniques to discover which technique results in the highest hit rate, dismiss rate, and f-score. We first describe the re-balancing techniques and model used, then present and analyze the results.

2. **RQ2. How classifiers perform compared with other prediction techniques?**

   This research question discuss the performance of classifiers after sampling the data compared with a stochastic genetic algorithm classifier and a deterministic classifier.

Chapter 4

Experimental Methodology

4.1 Assumptions

There are few assumptions, and justified prejudice in this work, which is scoped to the hardware/software tools, the paradigms for choosing Project for curating the dataset, *etc.*Some major assumptions for this work are as follows:

1. Nature of the Dataset

   The dataset being used is curated by earlier study[23], and we assume that they have reported the results accurately and correctly to the best of their knowledge.

2. Criteria for choosing SUT

   This work focuses on the early detection of performance regression introducing commit by performing performance test selection. Thus, the primary criterion for selected the System to test(project to perform static and dynamic analysis) is - the project should have performance tests written. The dependency on the programming language of the project or the programming principles it follows is out of the scope of this work.

3. Another assumption with performance tests for the work is that we assume that the tests are correct, and the flakiness of the criteria has been eliminated by the developers before running the study.

4. For classification and resampling techniques, we used Microsoft machine learning

studio [2]. Microsoft ML studio has algorithms and resampling techniques available as plug n play service. We assume that the algorithms and resampling techniques are accurate and correct in processing.

5. The previous study[23] used digital ocean VM's for running the performance tests as they are long-running and require dedicated resources. To reduce the noise that the VM's might have caused, each criterion for run five times on different dates and times to mitigate this noise.

4.2 Data collection

The dataset used in this work was developed by earlier study [23] paper, which is also used in the PRICE[5] paper. The authors uses *Git* project as the SUT (System Under Test). They describe the choice of Git as it is open source, and has built-in benchmarks for performance regression testing. Also, Git is an open-source project, and the commands are very familiar to us. The data was collected for 8798 commits originally. Some commits did not have proper tests because of which some of the commits were, discarded. This arises the need to drop some of these changes and make the dataset to 8956 commits.

After every commit, all the performance tests were run for the following two reasons:

- To test the tests and to mark the commit under the test as problematic.
- The other reason is to perform dynamic profiling in the code change and assess the results at runtime. Each of the tests is run at least five times to be sure that the results are robust, and there is no discrepancy in the results.
- To perform dynamic information, the authors used Linux perf[12]. The testing was performed by the module in python, known as the Lizard [1]. This module is used for static information gathering. The main idea behind this is to gather the Cyclomatic complexity, but it is used for other metrics as well. [2]

[1]http://terryyin.github.io/lizard/

[2]https://smilevo.github.io/price/

Table 4.1: Example of redundant rows in the dataset - Confusing for the classifier

Commit A	**Commit B**	**Benchmark**	**Del Func >= X**	**New Func >= X**	**Reached Del Func >= X**	**Top Chg by Call >= X%**	**Top >X% by Call Chg by >= 10%**	**Top Chg Len >= X%**	**Top Reached Chg Len >= X%**	**Hit/Dismiss**
8f449614	***12913a78***	***p4001-diff-no-index.sh-perf-report***	569	995	3	0.06	0.02	0	1	1
8f449614	***12913a78***	***p4211-line-log.sh-perf-report***	569	995	6	7.68	0.47	0	1	0

From the curated dataset from the earlier study, we removed the redundant rows for a commit pair, meaning that if for a benchmark the commit pair has been classified as a problematic commit, then we removed the other entries, which are classified as non-problematic. The rationale behind this is that if a commit pair has been classified as problematic for at least one benchmark, then it is introducing performance regression and should be treated as a problematic commit. As the table 4.1 shows that the commit pair is classified as problematic for *no-index.sh* benchmark and non-problematic for others. The other rows or benchmark entries are thus removed, essentially for this example, the 2nd row will be deleted.

As pointed in the table 4.1, the first two columns are the ids of the commits. To feed the dataset in the classifier, the id's were not relevant as we are not looking at the relevance of the commit ids to the project. Hence, we replace those columns with index columns, which gives us the number of the rows to 6353.

4.3 Approach

4.3.1 Resampling techniques

Random undersampling

The Random undersampling technique (RUS) is used in this work as one of the resampling techniquess to remove the imbalance in the dataset by randomly undersampling the majority class, in our case, it is the Dismiss class or the rows with commits as non-problematic. There has been a lot of work in the area of undersampling[25]. In our case, the resultant dataset was filtered to 812 entries in total with 400 positive class entries and 412 negative class entries. As the majorty class was downsized to 400 samples.

Random oversampling technique

The Random oversampling (ROS) is where we keep replicating the minority class until it is either equal or near to similar to the majority class. This approach is often criticized as the model is not exposed to any new data as the new entries are essential for the older ones. The cross-validation approach also does not eliminate this overfitting. [6].

SMOTE

Synthetic Minority Oversampling Technique(SMOTE) is a resampling technique that adds additional data points for the imbalanced class by generating data that is a variation of existing 1s within the dataset. SMOTE uses *k*-nearest minority class neighbor method to generate data for oversampling techniques [9].

In SMOTE, there are two variables that determine how the data is generated; one is G - number of samples from the original data and *k*; both of these can be specified by the user. The value of *k* is the number of nearest neighbors. The generation of similar examples approach in SMOTE by using the existing minority points, the synthetic samples are created nearing the samples of the minority class; it creates broader and less specific decision boundaries that empower the classifier with having a sense of generalization, and it increases their performance because it makes the classifier robust enough.

We approached the problem in three phases. The data collection phase is the first one that uses the historical data about the performance tests of the older commits to calculate the metric. The metrics values collected are with respect to the previous commit. This gives the chronological order for the dataset, and each commit pair consists of commits ordered in time. The metrics 1,2,6 in table 4.3.1 are the static metrics and the others are the dynamic metrics[5]. The static data is important to calculate benchmarks and allow the dynamic process to create them.

Table 4.2: 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.

4.4 Experimental setup

we Created experiments using azure ml studio [3] to generate a model to train a classifier for predicting performance regression introducing code changes. Since mentioning in Chapter 3, that the problem of training this classifier is a class imbalance problem; hence, we use the above mention in section 4.3.1.

[3]https://studio.azureml.net/

4.4.1 Training Algorithms

1. **Two class Boosted Decision Tree.** A boosted decision tree is an ensemble learning method in which the second tree corrects for the errors of the first tree; the third tree corrects for the errors of the first and second trees, and so forth. Predictions are based on the entire ensemble of trees together that makes the prediction.[4]. The implementation of Boosted Decision Tree in Azure is as follows:

   - The first step is an ensemble of weak learners
   - For each of the training sets, the transient out is the sum of all the outputs of the weak ensembles.
   - Calculating the gradient loss of each training set by the ensemble of weak learners.
   - The tree-building algorithm greedily selects the feature and threshold for which a split minimizes the squared loss about the gradient calculated above. The selection of the split is subject to a minimum number of training examples per leaf.

   This algorithm extends the tree with the final rule, and each leaf representing the value that was considered while coming up with that rule.

2. **Two Class Decision Forest.** The decision forest is based on the idea that instead of evaluating just one tree, it is better to have several trees and then create a general tree out of the rules of other trees.

   - The Decision Forest is a collection of trees [5] that are ordered by voting on the most output class.

[4]shorturl.at/jnT05

[5]https://docs.microsoft.com/en-us/azure/machine-learning/studio-module-reference/two-class-decision-forest

  - Each label in the trees of a decision forest is given probabilities, and based on these probabilities, the most output class is decided.
  - Each decision tree in a decision forest is given a Decision Confidence weight, and the tree with the highest decision confidence gets the maximum vote.

3. **Two class Support Vector Machines.** is a supervised machine learning model that is used for classification [6]. Essentially, the algorithm analyses the input data and recognizes the output in the multi-dimensional space called the hyperplane. For prediction, the SVM algorithm assigns new examples into one category or the other, mapping them into that same space.

4.5 Model

Since, there are two classes in the study which are of significance, which makes this a binary classification problem. To use the binary classification, we used the Microsoft Azure machine learning studio. Any model is formulated by the following steps:

1. The first step of the model is to import the dataset as in Figure A.1. Then based on the oversampling technique, we determine the next step. After determining the oversampling technique, we use the *split data* step to split the dataset into testing and training.

2. The training dataset is used to train the model using the respective training algorithm. The scoring model is to give the testing set labels by the trained model, which is further analyzed by the Evaluate model step. This step sums up the results by giving us the confusion matrix for that specific experiment.

3. In Azure ML studio, each step of the model is a block, and there are various techniques available for training. We focused on Two-class training techniques as in our

[6]https://docs.microsoft.com/en-us/azure/machine-learning/studio-module-reference/two-class-support-vector-machine

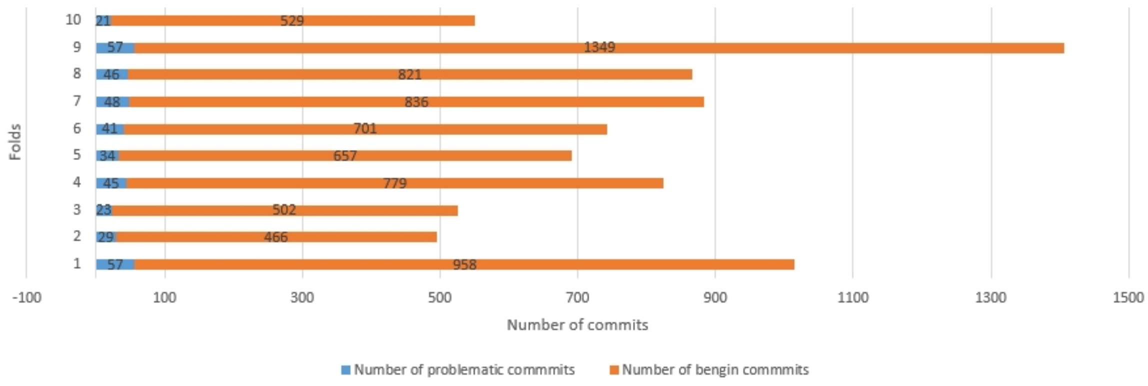

Figure 4.1: Number of problematic commits in each fold of the commit data set

Table 4.3: The distribution of folds

Fold	**Number of Hit's per fold**	**Fold Range**
1	57	1 - 634
2	29	635-1268
3	23	1269-1902
4	45	1903-2536
5	34	2537-3170
6	41	3171-3804
7	48	3805-4438
8	46	4439-5072
9	57	5073-5706
10	21	5707-6353

work; the classes are mainly two - Problematic/Non-problematic. Figure A.1 shows a model that was used to train the classifier using a boosted decision tree with the SMOTE+KNN oversampling technique.

The dataset we used had seven indicators 4.3.1, which are the static and dynamic metrics for the performance tests of the SUT. These indicators serve as the features for our classifier. Hence, we use the feature set of 7 for this study.

We experimented with three primary classifiers, mentioned in section 4.4.1. The next step after importing the dataset is to decide the oversampling methodology we wanted to take, which are mainly:

1. Chronological ordering of the dataset

2. Non-chronological ordering of the dataset.

Each of the techniques above mentioned had different ways to be tackled in the practical setup, which is as follows:

Chronological Ordering

1. **Resampling after splitting.** In this approach, we oversample our dataset only for the training fold and not the testing. This approach is to test how the classifier will perform when it is tested on an imbalanced dataset. When we apply the oversampling techniques after splitting the data into testing and training folds, it suffices the approach that the chronological is maintained as the oversampling techniques generate data close to the folds they are applied to in the training dataset, preserving the nature of the original data in the training set. The results of this approach are elaborated in Chapter 5 and the graphs depicting them (SVM) is Figure B.2.

Non-Chronological ordering

1. **Resampling at the start.** Once we have the dataset, we applied the rebalancing techniques even before splitting the dataset. The rebalancing techniques are discussed in section 4.3.1. When the dataset is rebalanced then we split the data into the ratio of 8:2, where 80% data is for the training set and the 20% in the testing set.

   The testing set is scored by the score model step in azure, using the trained model. Meanwhile, the training dataset is fed into the training the model for various techniques, as discussed in section 4.4.1 above. The tune hyper model parameters step tunes the metadata and the model and allow the range of features to be included in the model. This removes the outliers and enables the model to behave appropriately.

The model for this is set of experiment in Figure A.1 in Appendix A. The first step that is applied to the dataset is the oversampling technique SMOTE, which essentially does not add the data in the particular order and hence we lose the chronological order of the dataset. The model shows one classification algorithm, but, the other classification algorithms were used for training in the similar manner, by replacing the algorithm block with desired two-class classification algorithm

2. **Conventional 10-fold cross validation.** The normal cross-fold validation is on the lines of k-fold cross validation, where we used k=10 folds for the dataset. Each fold consists of approximately 635 entries with the number of problematic and non-problematic commits varying, as shown in figure 4.1. The data consists of 6353 commit pairs.

   In the approach, we split the in 9:1 ratio for the training and testing set, respectively. Hence, to be detailed, each run's testing set had 635 entries approximately, and the training set had 5700 entries. Each run made sure that each fold goes into testing at least once. Since there is diversification in the number of problematic commits in each fold, the result was scattered as well as shown in Figure B.5. This manner of experimentation doesn't consider the chronological order of the commits.

   In the dataset, the commit pairs are ordered chronologically. This approach is taken into consideration to test of the time of the commits have an impact on the classifier. A better example can be that one can assume that the older commits will add value to the newer commit prediction.

Chapter 5

Experimental Results and Evaluation

5.1 RQ1. Which sampling technique provides the best representation for data classes?

To address this research question, we used sampling techniques mentioned in 4.3.1 since each technique had different approaches to rebalancing. The True Hit rate and True Dismiss rate were focussed points of evaluation, looking from Perphecy and PRICE papers. The true hit rate is calculated as follows:

The baseline for our experiments is termed as the original technique, where we do not use any oversampling technique in the dataset. This allows us to see the drawbacks or impact of the imbalanced data on the classifier. Over the training folds, we observed that the baseline did poorly for Hit rate even for all the folds. Thus, it is necessary to perform oversampling for the classifier to predict performance regression inducing changes.

Table 5.1: Confusion matrix mapping to the graphs

True Positive	Correctly Classified as Problematic commits
True Negative	Correctly classified as non-problematic commits
False Positive	Incorrectly classified as problematic commits
False Negative	Incorrectly classified as non-problematic commits

In this study, we focus on the following statistics and use these statistics to compare with each other. The True hit rate is the rate of classification of problematic commits in the testing dataset. We look at this metric is important because we care about the problematic commit to be correctly identified even if there is a non-problematic commit that is incorrectly identified. This trade-off is considered looking at that the problematic commits need to be identified in totality. Some of the key terms relevant to our study are elaborated in Table 5.1.

Based on the above legend, the statistic measures that we used to evlauate our classifier are as follows:

- True Hit Rate (Recall):
$$TrueHitRate = \frac{\text{TP}}{\text{TP+ FN}}$$
- True Dismiss Rate:
$$TrueDismissRate = \frac{\text{FP}}{\text{FP+ TN}}$$
- F-Score:
$$F_1 = \frac{2\times\text{Precision}\times\text{Recall}}{\text{Precision}+\text{Recall}}$$
- Precision:
$$Precision = \frac{\text{TP}}{\text{TP + FP}}$$
- False Negative Rate:
$$FalseNegativeRate = \frac{\text{FN}}{\text{FN+ TP}}$$

5.1.1 Chronological Oversampling.

As mentioned in section 4.5, the chronological order is used to train the model on the older commits and test on the newer commits. The rationale behind this was that the metrics for the older commits would train the classifier to predict the newer commits as problematic or non-problematic. Since there are 10 folds, so necessarily, the 10th fold can be put to testing and use the folds 1-9 for training. Next, we used 1-8 folds for training and test fold 9 and 10 combined. This goes until we reach fold 1-5 in training and folds 6-10 in testing.

Table 5.2: Chronological Fold Test Validation

Training Algorithm	Rebalancing Technique	True Hit Rate	True Dismiss Rate	F-Score	Precision	False Negative Rate
Boosted Decision Tree	RUS	0.1787453906	0.1821146471	0.1780875442	0.04498921976	0.8212546094
	SMOTE	0.2243903493	0.1545033568	0.1762800788	0.0703470579	0.7756096507
	Original	0.1009920972	0.03782789823	0.05005803906	0.139391629	0.8990079028
	ROS	0.1696790569	0.1484521469	0.1447484415	0.2656874391	0.8303209431
Decision Forest	RUS	0.4420084243	0.6445126614	0.4355417776	0.09148237414	0.5579915757
	SMOTE	0.1438335458	0.1324588977	0.1358509325	0.06309506482	0.8561664542
	Original	0.04543744665	0.01447838174	0.0202164366	0.3266321045	0.9545625533
	ROS	0.09701508917	0.1376588294	0.1037918564	0.2388171226	0.9029849108
SVM	RUS	0.3421812755	0.7088100495	0.2809741504	0.1051010537	0.6578187245
	SMOTE	0.4790627641	0.5839051037	0.4586878121	0.08509873535	0.5209372359
	Original	0	1	0	0	1
	ROS	0.6323980198	0.3477920738	0.3884175701	0.08119802695	0.3676019802

Table 5.3: Rebalance whole dataset, chronological ordering

Training Algorithm	Rebalancing Technique	True Hit Rate	True Dismiss Rate	F-Score	Precision	False Negative Rate
Decision Forest	RUS	0.579	0.569	0.478	0.52	0.421

We did not see a stark difference in using this approach as in the Appendix B table 5.2 and which is why we cannot explicitly say that this way of testing is better or the chronology of the commits makes a difference as this can be true because the assumption that the nature of the changes will be the same is not valid. Over time, the style of changes might be identical, but the code is evolving, and thus the changes will be different, hence the underlying metrics.

5.1.2 Non-Chronological Oversampling

Conventional 10-fold cross validation

This approach of oversampling allowed us to use conventional 10-fold cross validation. In this, we saw that the classifier performed better than the chronological order. The rationale for this is the nature of the feature set and that each fold had different results. We ran each of the oversampling technique with all the Training algorithms and compared them with the chronological as well as the resample at the start technique 5.5.

Table 5.4: Conventional 10 fold cross validation

Training Algorithm	Resampling Technique	True Hit Rate	True Dismiss Rate	F-Score	Precision	False Negative Rate
Boosted Decision Tree	RUS	0.45	0.7012734179	0.4753089314	0.09241315504	0.55
	SMOTE	0.1794	0.8612413559	0.2845435054	0.08070841639	0.8206
	Original	0.1419	0.9305637052	0	0.1431895949	1
	ROS	0.2875206	0.7694887097	0.3711328764	0.1176883041	0.7124794
Decision Forest	RUS	0.443	0.69	0.501	0.098	0.557
	SMOTE	0.2472482277	0.7682486997	0.3294832703	0.08572634358	0.7527517723
	Original	0.1040007812	0.9619621157	0.178143959	0.2458295497	1
	ROS	0.316	0.782	0.414	0.207	0.684
SVM	RUS	0.608	0.504	0.483	0.078	0.392
	SMOTE	0.6359261497	0.5247393608	0.4760993039	0.07439706062	0.3640738503
	Original	0	1	0	0	1
	ROS	0.43	0.637	0.361	0.062	0.57

There is no specific reason or rationale behind each technique's behavior for the classifier. Still, we are sure that undersampling is the best among all along with undersampling; the data that is present is rebalanced, and along with that, it is accurate. As using RUS, we remove the extra zeroes, hence the dataset is shortened to 800 rows approximately, out of which the training and testing are curated. Theoretically, this will give the best results as this data is the most realistic and close enough representation of the real data. Here the Dismiss rows (majority class) are under-sampled randomly, which removes any concentrated feature set values. Hence, we get the recall of .608, which is close to SMOTE, which uses K-nearest neighbor to generate the data. SMOTE+KNN can produce data as it uses the existing data to create more data, and it lies on the same lines of RUS under the hood.

The key things that we can see:

- The chronological order does not affect the oversampling of the data, and the classifier is also not affected by commits order.
- The Problematic and non-problematic commits are varied to a great extent among the folds. If we look at the table 4.3, the number of problematic commits in each fold

Table 5.5: Rebalance at the start experiment results

Training Algo	Rebalancing Tech.	Recall	True Dismiss rate	F-score	Precision	False Negative Rate
Boosted Decision Tree	RUS	0.8	0.7228915663	0.7594936709	0.7356321839	0.2
	SMOTE	0.8788958148	0.8445378151	0.860130719	0.8421501706	0.1211041852
	ROS	0.9980525803	0.9714285714	0.9845606204	0.9678942398	0.001947419669
	Original	0.3125	0.9831932773	0.4742602351	0.5555555556	0.6875
Decision Forest	RUS	0.75	0.7710843373	0.7603960396	0.7594936709	0.25
	SMOTE	0.8539626002	0.8092436975	0.8306626245	0.8086003373	0.1460373998
	ROS	1	0.9756302521	0.9876648235	0.9725378788	0
	Original	0.1625	0.9756302521	0.2785971389	0.3095238095	0.8375
SVM	RUS	0.6625	0.5542168675	0.6235294118	0.5888888889	0.3375
	SMOTE	0.6835541699	0.4798319328	0.5638560023	0.5862299465	0.3164458301
	ROS	0.4391431353	0.7	0.5397042482	0.5581683168	0.5608568647
	Original	0	1	0	0	1

is different from that in the other folds. For example, in the first and the last fold, this number is much lower than the others.

The training algorithms that we used for this approach are mentioned in section 4.4.1. Analyzing the results(Table 5.4) we can observe that the testing data being imbalanced, causes the True Hit rate to be shallow and True Dismiss rate to be high when we don't use any oversampling technique.

5.2 RQ2. How classifiers perform compared with other prediction techniques?

In this research question, we look at the past techniques for the classifier and how they perform with the dataset. Alshoaibi *et al.*[5] uses Evolutionary algorithms like NSGA-II for the training of the classifier for the same problem. The essence and comparison of the best of different classifiers are shown in Figure 5.6.

Among all the experiments we performed, the best results, SMOTE, when applied to the whole dataset, before splitting gives a high true positive rate. SMOTE+KNN generates the closest minority results, feeding new data to the classifier. We consider the possibility of over-optimism of the model as the testing dataset is also a split from the aggregate data, which might not be the case in real scenarios.

Table 5.6: Performance of approaches under comparison in terms of Average (TPD,TDR,F-Score)

Approach	Average TPR	Average TDR	Average F-Score
KNN	0.04	**0.98**	0.09
NSGA-II	0.59	0.63	**0.60**
Perphecy	0.55	0.48	0.51
Our Results	**0.63**	0.55	0.47

Binary Problem and Rule for prediction

Since we used the Azure machine learning studio to perform the machine learning tasks, it shows all the trees that it used but does not the final tree that was generated. Also, for boosted decision tree and Decision Forest, there is no single tree that serves as the rule for making predictions. The model is illustrated in figure A.1 to show the steps that are used, and these steps are rearranged for different experiments, but primarily the steps can be treated as independent blocks that can be used in the desired order to create the desired model for testing.

The rationale behind selecting SMOTE with SVM as the better classification technique is that in ROS, the oversampling is carried out using replication of the minority, which is not a good measure as it not a representation of a realistic scenario. Over-optimism refers to the phenomenon when the classifier is fed with replicas or the same pattern in the testing and training dataset. This makes the classifier biased to that specific pattern and which is why the oversampling technique using SMOTE at the start might lead to over-optimism in a practical setting.

Chapter 6

Threats to Validity

In this chapter, we highlight the problems that this study might face in a practical setting. These threats to validity are mainly described in the following manner [39].

6.1 Internal Validity

The internal threats for this study concerning the dataset are similar to the on, present in the Perphey paper[23].

The ordering of the commits not necessarily mean that they depict the real time-ordering. The project used for curating the dataset is git, which uses git for version control. In git, there are a lot of ways, one of which is the branch control. This discrepancy does not affect our model prediction because if even the commits are not directly related, their time chronology is maintained.

Neither this study or the previous study takes into account the change or accumulation of changes in time. This essentially means that if the commit A and commit B are not directly related, and there are commits in between, this study does not look at that impact. As mentioned in Perphecy paper, the more viable way to do this will be to see the accumulation of change in a case the commit is seen as problematic.

The classifier model built on this study, and different techniques have various uses. The rebalance at the start technique gives promising results. Still, as explained in [38], this approach can prove to be over-optimistic, and overfitting as this can skew the real representation of the scenario. The practice of using oversampled data for testing some times mocks the actual situation. In our study, this does not affect a lot like the data that is generated by SMOTE, which used KNN to create data that is close to the current feature vectors.

6.2 Construct Validity

In this section we talk about the threats of study to the real-world applications.

Since the dataset is curated using git as the source code. This code is platform-independent, and hence the dataset, the set of tests is not enough for curating the performance metrics. The better way of doing this is to avoid continuous testing for all the commits, as, in a practical application, this is not feasible. It is better to have a buffer to maintain the results of previous tests and start with that buffer when there is a newer commit.

Since the performance tests (benchmarks) for the project (*git*) were run, and each commit pair had several reports generated. If even one of the tests comes positive in introducing regression changes, then practically it is Hit. The dataset had multiple instances of the same commit pair where the benchmarks are set to negative or non-problematic. This was mitigated by eliminating the confusing entries from the dataset for a set of commits.

The Boosted Decision Tree training algorithm sometimes creates overfitting and over-optimistic as it tunes the hyperparameters and with each tree generated. This tuning can be considered to be skewing the hyperparameters, and it sometimes creates a healthy tree that might be unrealistic in some scenarios.

6.3 External Validity

The project used for building the Machine Learning Classifier is based on one project. This might not be a strong premise for the classifier if put in a practical [32]. The classifier needs to be trained on several projects so that it is robust about the features and has more data to be trained on. In order to mitigate this, we used the testing and training of the same dataset but SMOTE, mocked for us the data generation, and that made sure that the data is close enough to the original data set.

Another threat to the dataset is that the classifier might be skewed because of the severe imbalance in the practical setup. In a functional structure, if the oversampling techniques are not chosen with care, then it can skew the classifier.

Chapter 7

Conclusion & Future Work

This work aims at creating a classifier that can perform early-prediction of the Performance regression introducing code changes. We compare with the state-of-the-art approaches and see the impact of static and dynamic metrics for training the classifier. We observe in this study that the classifier selection is subjective to the imbalanced data and the problem of class imbalance. The results are compared with different oversampling techniques, and each method is used with varying algorithms of training.

The work done in the previous paper does not tackle all the oversampling techniques for this problem. This study gives a broader scope by looking at the chronological order of the commits as well. The survey can be extended by using other metrics like Flux and Cyclomatic Complexity as they are indicators of the quality of the source code.

We plan to extend this study uses one Project(*Git*) for data collection, which is not sufficient for the realistic setting and to make the classifier robust enough to be applied in a realistic developer scenario. The project constraint is just that the project should have formal testing. We plan on extending the feature ranking portion in order to better understand the impact of each feature on the Classification of Problematic commits. The feature space is adequate but has scope for extension.

This study can be expanded by looking at the different datasets of the same genre of testing, namely, performance tests.

Bibliography

[1] Lkp. `https://01.org/lkp/`.

[2] Microsoft. `https://studio.azureml.net/`.

[3] Talos. `https://wiki.mozilla.org/Buildbot/Talos`, 2014.

[4] 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.

[5] Deema ALShoaibi. Characterizing performance regression introducing code changes. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 634–638. IEEE, 2019.

[6] Gustavo EAPA Batista, Ronaldo C Prati, and Maria Carolina Monard. A study of the behavior of several methods for balancing machine learning training data. *ACM SIGKDD explorations newsletter*, 6(1):20–29, 2004.

[7] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. Are test smells really harmful? an empirical study. *Empirical Software Engineering*, 20(4):1052–1094, 2015.

[8] U. Bhowan, M. Johnston, M. Zhang, and X. Yao. Evolving diverse ensembles using genetic programming for classification with unbalanced data. *IEEE Transactions on Evolutionary Computation*, 17(3):368–386, 2013.

[9] 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.

[10] Nitesh V. Chawla, Nathalie Japkowicz, and Aleksander Kotcz. Editorial: Special issue on learning from imbalanced data sets. *SIGKDD Explor. Newsl.*, 6(1):1–6, June 2004.

[11] Tim Chen, Leonid I Ananiev, and Alexander V Tikhonov. Keeping kernel performance from regressions. In *Linux Symposium*, volume 1, pages 93–102, 2007.

[12] Arnaldo Carvalho De Melo. The new linux'perf'tools. In *Slides from Linux Kongress*, volume 18, 2010.

[13] A. B. De Oliveira, S. Fischmeister, A. Diwan, M. Hauswirth, and P. 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, 2017.

[14] 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.

[15] Augusto Born de Oliveira, Jean-Christophe Petkovich, Thomas Reidemeister, and Sebastian Fischmeister. Datamill: Rigorous performance evaluation made easy. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ICPE '13, page 137–148, New York, NY, USA, 2013. Association for Computing Machinery.

[16] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and Tanaka Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-ii.

In *International conference on parallel problem solving from nature*, pages 849–858. Springer, 2000.

[17] 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.

[18] Vaishali Ganganwar. An overview of classification algorithms for imbalanced datasets. *International Journal of Emerging Technology and Advanced Engineering*, 2(4):42–47, 2012.

[19] Shadi Ghaith, Miao Wang, Philip Perry, and John Murphy. Profile-based, load-independent anomaly detection and analysis in performance regression testing of software systems. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 379–383. IEEE, 2013.

[20] M. Gligoric, L. Eloussi, and D. Marinov. Ekstazi: Lightweight test selection. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 713–716, 2015.

[21] Xinjian Guo, Yilong Yin, Cailing Dong, Gongping Yang, and Guangtong Zhou. On the class imbalance problem. In *2008 Fourth international conference on natural computation*, volume 4, pages 192–201. IEEE, 2008.

[22] Guo Haixiang, Li Yijing, Jennifer Shang, Gu Mingyun, Huang Yuanyue, and Gong Bing. Learning from class-imbalanced data: Review of methods and applications. *Expert Systems with Applications*, 73:220–239, 2017.

[23] Kevin Hannigan. An empirical evaluation of the indicators for performance regression test selection. 2018.

[24] Haibo He and Edwardo A Garcia. Learning from imbalanced data. *IEEE Transactions on knowledge and data engineering*, 21(9):1263–1284, 2009.

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

[26] Tomas Kalibera, Lubomir Bulej, and Petr Tuma. Automated detection of performance regressions: The mono experience. In *13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 183–190. IEEE, 2005.

[27] Tomas Kalibera, Jakub Lehotsky, David Majda, Branislav Repcek, Michal Tomcanyi, Antonin Tomecek, Petr Tuma, and Jaroslav Urban. Automated benchmarking and analysis tool. In *Proceedings of the 1st International Conference on Performance Evaluation Methodolgies and Tools*, valuetools '06, page 5–es, New York, NY, USA, 2006. Association for Computing Machinery.

[28] Victoria López, Alberto Fernández, Jose G Moreno-Torres, and Francisco Herrera. Analysis of preprocessing vs. cost-sensitive learning for imbalanced classification. open problems on intrinsic data characteristics. *Expert Systems with Applications*, 39(7):6585–6608, 2012.

[29] Qi Luo, Denys Poshyvanyk, and Mark Grechanik. Mining performance regression inducing code changes in evolving software. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 25–36. IEEE, 2016.

[30] Tomasz Maciejewski and Jerzy Stefanowski. Local neighbourhood extension of smote for mining imbalanced data. In *2011 IEEE Symposium on Computational Intelligence and Data Mining (CIDM)*, pages 104–111. IEEE, 2011.

[31] R Mollineda, R Alejo, and J Sotoca. The class imbalance problem in pattern classification and learning. In *II Congreso Espanol de Informática (CEDI 2007). ISBN*, pages 978–84, 2007.

[32] Nuthan Munaiah, Casey Klimkowsky, Shannon McRae, Adam Blaine, Samuel A Malachowsky, Cesar Perez, and Daniel E Krutz. Darwin: a static analysis dataset

of malicious and benign android apps. In *Proceedings of the International Workshop on App Market Analytics*, pages 26–29, 2016.

[33] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, page 265–276, New York, NY, USA, 2009. Association for Computing Machinery.

[34] Ketil Oppedal, Kjersti Engan, Trygve Eftestøl, Mona Beyer, and Dag Aarsland. Classifying alzheimer's disease, lewy body dementia, and normal controls using 3d texture analysis in magnetic resonance images. *Biomedical Signal Processing and Control*, 33:19–29, 2017.

[35] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering*, 23(3):1188–1221, 2018.

[36] 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.

[37] Juan Pablo Sandoval Alcocer, Alexandre Bergel, and Marco Tulio Valente. Learning from source code history to identify performance failures. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ICPE '16, page 37–48, New York, NY, USA, 2016. Association for Computing Machinery.

[38] Miriam Seoane Santos, Jastin Pompeu Soares, Pedro Henrigues Abreu, Helder

Araujo, and Joao Santos. Cross-validation for imbalanced datasets: Avoiding overoptimistic and overfitting approaches [research frontier]. *ieee ComputatioNal iNtelligeNCe magaziNe*, 13(4):59–76, 2018.

[39] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

Appendix A

Azure experimental model

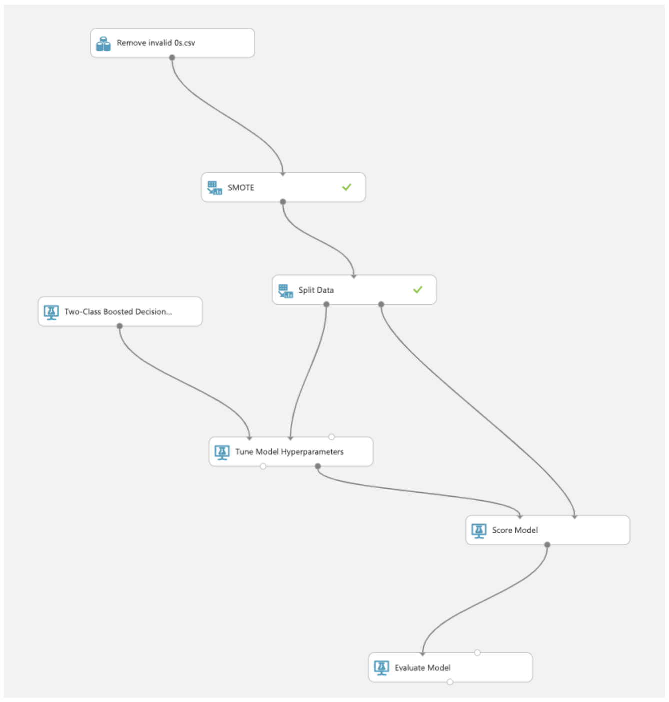

Figure A.1: Azure Classifier model for SMOTE+KNN with Boosted Decision Tree

Appendix B

All Graphs for the experiments

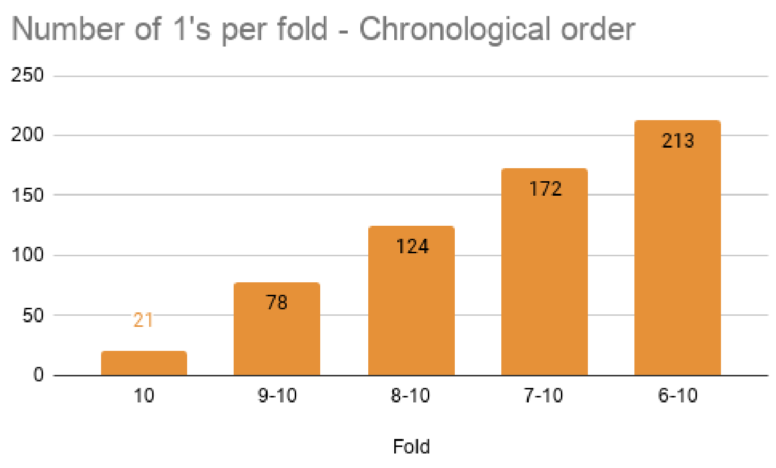

Figure B.1: Number of Hit’s per fold - Chronological order

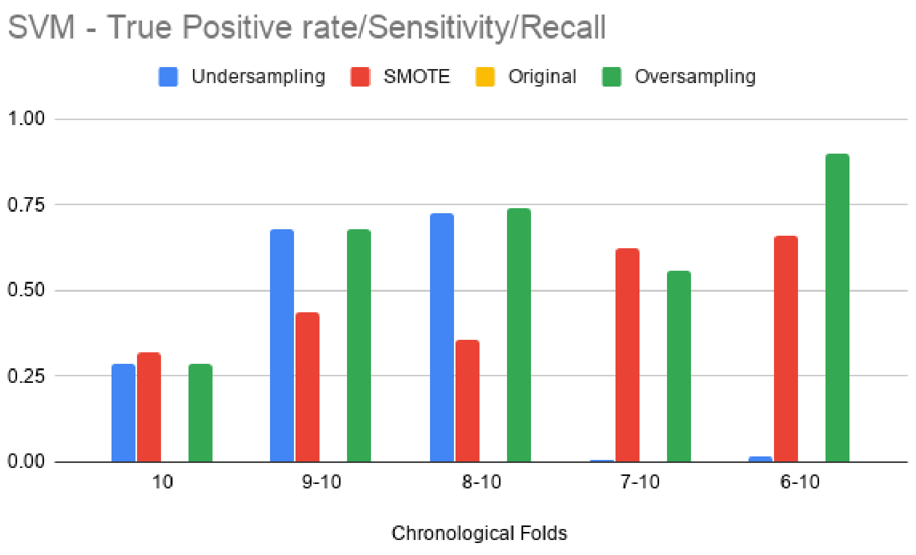

Figure B.2: Support Vector machine True postive rate with resampling techniques - 10 fold cross Validation

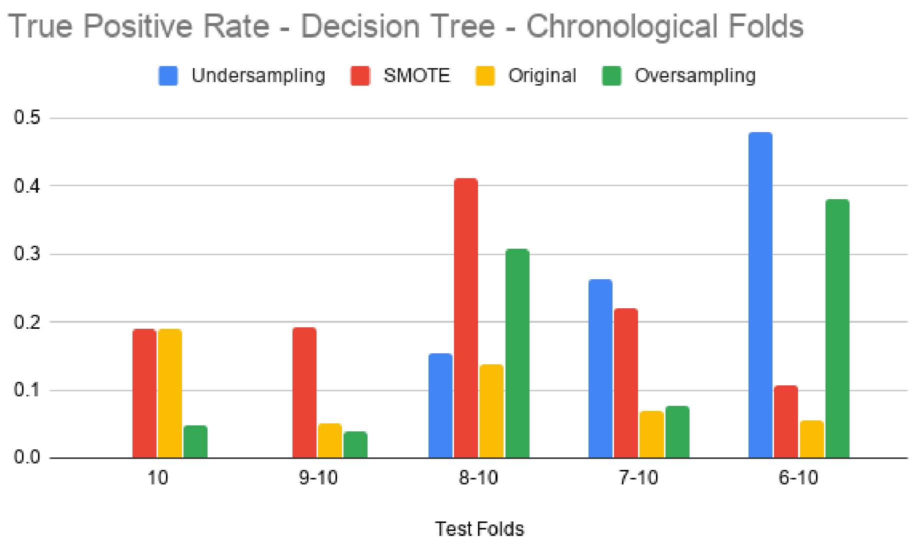

Figure B.3: True hit rate for Boosted Decision Tree in Chronological fold ordering

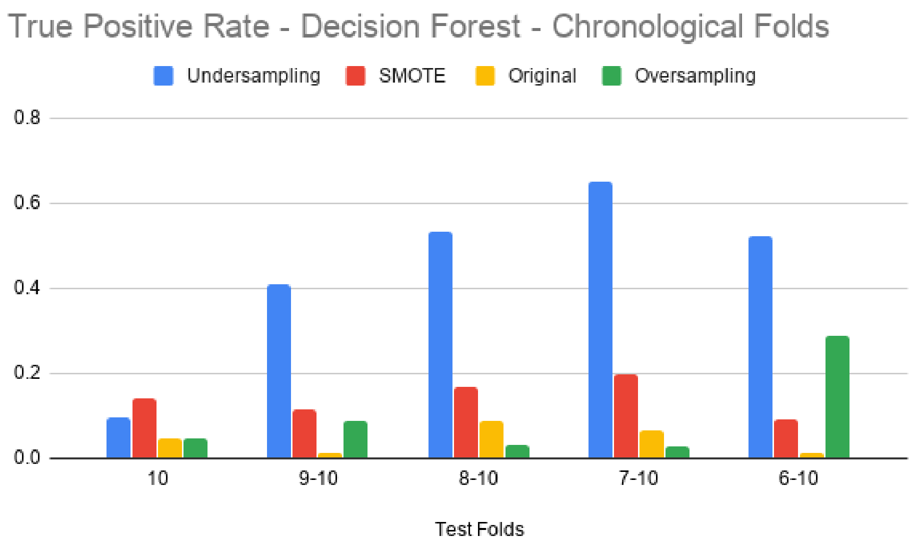

Figure B.4: True hit rate for Decision Forest in Chronological fold ordering

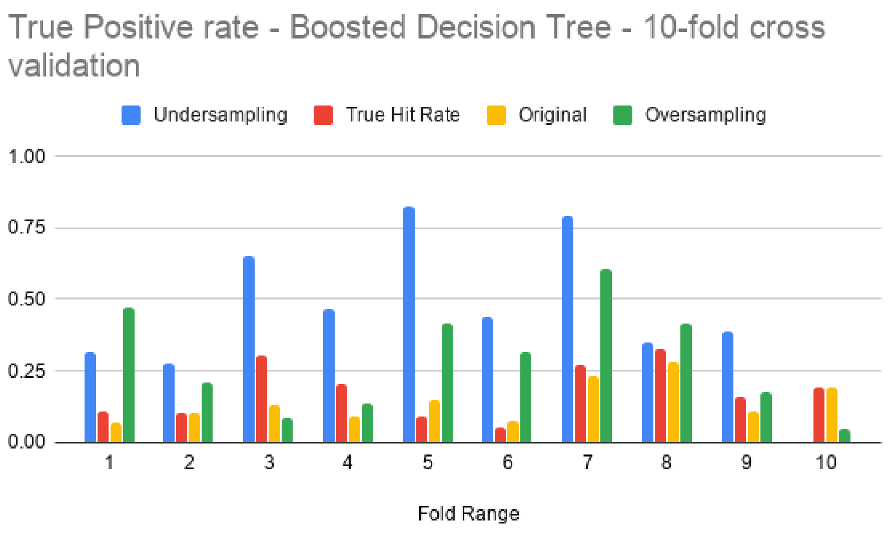

Figure B.5: True hit rate for Boosted Decision Tree using 10 fold cross validation