Understanding and Predicting Method-level Source Code Changes Using Commit History Data

by

Joseph Heron

A thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science

in

Computer Science

University of Ontario Institute of Technology

Supervisor: Dr. Jeremy Bradbury

October 2016

Abstract

Software development and software maintenance require a large amount of source code changes to be made to a software repositories. Any change to a repository can introduce new resource needs which will cost more time and money to the repository owners. Therefore it is useful to predict future code changes in an effort to help determine and allocate resources. We are proposing a technique that will predict whether elements within a repository will change in the near future given the development history of the repository. The development history is collected from source code management tools such as GitHub and stored local in a PostgreSQL. The predictions are developed using the machine learning approaches Support Vector Machine and Random Forest. Furthermore, we will investigate what factors have the most impact on the performance of predicting using either Support Vector Machines or Random Forest with future code changes using commit history. Visualizations were used as part of the approach to gain a deeper understanding of each repository prior to making predictions. To validate the results we analyzed open source Java software repositories including; acra, storm, fresco, dagger, and deeplearning4j.

Acknowledgments

I would first like to thank my thesis advisor Dr. Bradbury of the Computer Science Faculty at the University of Ontario Institute of Technology. Dr. Bradbury was always available to provide direction and feedback about both research and writing.

I would also like to thank the Dr. Szlichta for his thoughtful advice and recommendations throughout the work on this thesis.

To my colleagues at the University of Ontario Institute of Technology and lab mates in the Software Quality Research Lab, thank you for the stimulating discussions throughout the last two years.

Finally, I must express my very profound gratitude to my family and friends for providing me with unfailing support and continuous encouragement throughout my years of study, research and writing. This accomplishment would not have been possible without them. Thank you.

Contents

List of Figures

List of Tables

Abbreviations

AI Artificial Intelligence.

API Application Programming Interface.

DVCS Distributed Version Control System.

IR Information Retrieval.

LD Levenshtein Distance.

MSR Mining Software Repositories.

NLD Normalized Levenshtein Distance.

OSS Open Source Software.

RF Random Forest.

SQL Structured Query Language.

SVM Support Vector Machine.

SVN Apache Subversion.

SWR Sample Window Range.

VCM Version Control Management.

VCS Version Control System.

Chapter 1

Introduction

Software has become pervasive and integrated with numerous platforms and applications such as mobile devices, web sites, embedded systems, safety critical systems. Creating and managing a software application can be time consuming and resource intensive. The development of software applications commonly integrate the usage of Version Control System (VCS) to manage the application by storing the current version as well as previous versions in a repository. The development of a repository is limited by the resources available to the team developing the application. Effective allocation of these limited resources could be the deciding factor in whether repository will be successful or not. Furthermore, a VCS manager such as GitHub could leverage repository resource allocation for managing the resources provided to repositories hosted by them.

Previous works in change prediction include Bantelay, Zanjani and Kagdiw who studied evolutionary coupling to create predictions of commit and interactions within a repository [3]. Giger, Pinzger and Gall analyze an approach for predicting the type of change that will occur [14]. Hassan and Holt predict change propitiation of a purposed change [21]. Kagdi and Maletic outline an approach for predicting software

changes through a dependency analysis of the repository history [25]. Ying, Murphy, Ng and Chu-Carroll create a method for predicting source code changes given a change within the repository [45]. The predictions are made by identifying elements within the repository that are associated through frequent co-changes. We hope to build on previous research to provide change predictions to help improve resource allocation for both repository developers as well as VCS managers.

1.1 Objective & Methodology

The mining of open source software repositories is widely used to help research into various software topics relating to software development and quality assurance. Research can provide improvements to the development process of software repositories. With an improved development process, more repositories may succeed in accomplishing their outlined goal. The process of developing a repository will of course take time to complete. The time for a repository to be completed relies on numerous factors including scope, man power, experience. Over the course of development, changes will be made to repository. Changes can be made to almost any part of the repository including design, number of developers and type of developers. These changes will in most cases have a measurable impact on the repository. In case of adding more developers, the intended result may be to increase functional capabilities within a shorter span of time. Even with an intended result, the actual result may differ and should be measured to determine the effectiveness of a given change.

Development of a software application will attempt to solve any number of problems. A few examples of typical software problems are; video games, telecommunication or financial. The success of an application however does not rely on a single factor. Software applications may fail in many different ways affecting the scope,

cost, or timeliness of the project. Furthermore, software development often continues long after the product is delivered to clients. Managers or developers may decide to increase the resources available for the development in an attempt to solve the problem. Without clear understanding or knowledge of what resources are necessary can exacerbate the problem. For example, allocating more developers to work on a project will also require more coordination between developers which can increase overhead. Providing predictions of upcoming code changes can provide developers and managers insight into the development schedule of their application and help them make more informed decisions regarding future resource allocation.

The developers of the repository should manage the growth of the repository to ensure that the changes that are made result in an expected outcome. Keeping track of every change to a repository can be difficult because of external changes which are beyond the control of the developers. However, for the majority of the changes within the repository they are kept track through VCS. With proper use of a VCS, the important changes made to the repository available. This can help keep previous releases of the software available or even help resolve a bug that was introduced in a recent change. Furthermore, developers have control over what is stored in the VCS allowing for granularity based on developer preferences. With numerous developers, a VCS can also help improve how these developers interact and share the changes they made. Some commonly used VCS include Git [1], Apache Subversion (SVN)[2] and Mercurial[3].

The impact of changes can be measured and provides insights into how the repository changes. However first the data must be collected, processed and stored. While changes may occur in various forms, a more accessible one would be the source code

[1] `https://git-scm.com/`
[2] `https://subversion.apache.org/`
[3] `https://www.mercurial-scm.org/`

changes in the software repository. These changes are very fine grain since they will account for almost all functionality changes with the repository. The only functionality changes not accounted by source code would be external changes (e.g. library changes). Changes will map to functionality changes that provide fixes, new functionality, or removal of functionality. The source code changes will provide a large amount of noise since every change is included. This excessive granularity can make tracking the desired changes more difficult. Visualization of the data collected allows for a more accessible look at the data to provide potential insights.

As discussed earlier, there are two main types of software repositories that are developed, either closed source or open source. Open Source Software (OSS) repositories will generally provide access to the source code, the ability to change and finally redistribute the changes. OSS is widely used in developing software repositories of various sizes and scope. In these repositories developers are able to contribute towards the a repository that is often used by a wider audience. While smaller OSS repositories may have a small number of developers, larger repositories can contain developers from numerous locations around the world contributing at different times. The development of OSS is often the focus of research related to software development since the repositories are open and freely available. The authors are able to publish and use the data as they wish since it is publicly available. There are also countless OSS repositories to study and investigate.

The collection of data is done through data mining. Data mining is the act of collecting data from one or more sources to use for another goal. Often data mining will use a data source not traditionally used, since the goal of data mining is to extract and use information. The actual use of the data once collected can vary greatly from visualizing to modeling. Data can also be collected in several forms including continuous streams of data, sporadic data and one time collection. Depending on

what type of data is being collected and the purpose of the collection the means of collection may also vary. Another concern related to data mining is that of Big Data. If a source provides a wealth of data, then extra measures should be taken to manage the size of the data set. Without diligent management, a data set can become unwieldy with massive overhead that are entirely avoidable.

The goal of the approach is to predict changes that will occur within the repository using the commit history. In this case machine learning techniques are leveraged to create a model based on the data collected through mining GitHub to predict changes. Machine learning techniques are widely used to support the completion of difficult tasks that involve patterns. A machine learning algorithm is generally an algorithm that attempts to detect and mimic patterns within a data set. There are many different types machine learning algorithms including:

1. Support Vector Machine (SVM)
2. Random Forest (RF)
3. Artificial Neural Network (ANN)
4. Deeplearning Artificial Neural Network (ANN)
5. Regressors
6. Bayes Naïve Classifiers (BNC)

Each technique provides advantages and disadvantages depending on the purpose and the data set in use. The primary focus will be on Support Vector Machine (SVM) and Random Forest (RF) since they are used as part of the proposed work. These two machine learning algorithms were selected because of their wide use in existing works. Specifically, research related to data mining often employed these techniques

to make use of the collected data. Regardless of the predictive model, the training and testing layout is the same, by sampling a subset from the data source to use towards training of the model. To test the model a second subset that is distinct from the first is retrieved to permit the performance of the model to be measure.

SVM is an algorithm that attempts to classify data into two distinct categories. This algorithm is a supervised learning technique that requires a training data set to build the categorization model. The training set will consist of data samples from each classification as well as which classification the data sample belongs to. After creating the model for a SVM new data vectors can be provided to the model and be classified into one of the two categories. The model will be constructed by attempting to linearly separate the data into two distinct groups. If the data cannot be separated linearly, then the data is mapped to a higher dimension to allow for proper separation. During the separation of data points into two groups, the model may reclassify data points in an attempt to fix errors within the data set. This feature allows for some error to be present within the training set without causing further errors. In the case that points which are valid are detected as errors then the separation has generated errors and the features or data used may not be useful towards making a prediction.

RF is another supervised learning technique that requires a training data set to create an prediction model. The origin of random forests is the decision tree learning method. A single decision tree creates a tree structure where each internal node in the tree represents a decision where in the final destination is the outcome. RF extends decision trees to address the tendency for a decision tree to overfit the data. A RF uses numerous decision trees as well as a modified version of bootstrap aggregation to get more robust predictions.

The machine learning algorithm makes use of the training data to create a model for predicting the classification of a given value. An analysis of change data can help

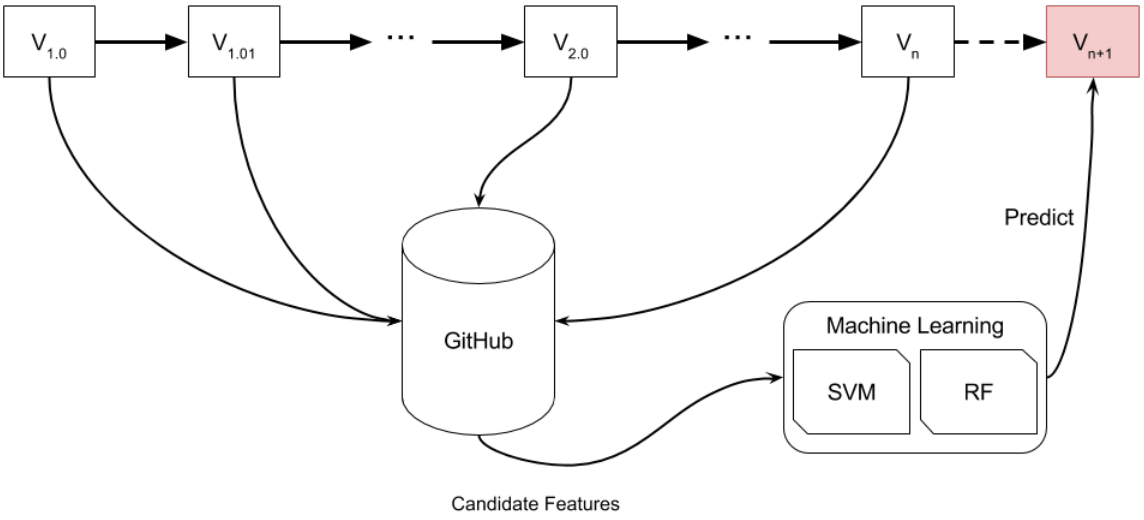


Figure 1.1: Approach Overview

in the selection of useful information for creating the predictive model. The data is collected from the repository's historical data that can provide a large data set to work with. Managing the data and selecting ideal samples for training must be considered to provide a strong predictive model.

We propose a tool that assists in managing the development of software repositories by predicting changes that are likely to occur. This work explores leveraging change prediction of the source code using the commit history to assist in the development of OSS scale repositories. The key factors; sampling size, feature set and data balancing are investigated using the tool to provide a deeper understanding the feasibility of the tool. Several OSS repositories were selected to conduct experiments to determine the impact of each of the factors.

1.2 Contributions

Our contributions are in mining of OSS, visualization of a repository's change history, machine learning change prediction, data collect which can be used and extended. Providing clear and accessible visualizations allows for the development data to be inspected more thoroughly. Likewise, strong predictions of change can assist in the development of more robust, efficient and less costly software programs. Finally, the collection of historical data for use of predicting future changes is presented as a possible option when predicting future changes within a repository.

There are several different areas where this work can be applied and provide improvements. The main area which this research is applicable would be that of software development by providing support during the development process. The prediction of future changes within a repository are made available to developers to support them when choosing tasks or making new changes to the repository. With the knowledge of where changes are likely to occur within the repository developers may be more prepared in making such changes. This is especially true if this work was extended to provide more specific change information about future changes. Another potential use for this approach would be in resource allocation for software repository development. A larger repository with numerous developers contributing will require each developer to work on various task an attempt to limit conflicting contributions. With the ability to predict where future changes will likely occur developers can coordinate more effectively with other developers mange interactions and overlap. Finally, the approach could help with resource allocation for a VCS provider, such as GitHub. A VCS could use the repository data to efficiently allocate resources based change predictions for each repository. If a repository is likely of have near future changes then more resources are necessary for that repository compared to one that

is less likely to receive changes in the near future. With strong predictions a system could be more effectively managed and offer savings to the company.

1.3 Organization

The remainder of the thesis is organized into 5 more chapters.

1. Literature Review which provides more details related to the foundation of this work. Primarily this chapter will cover the data that is collected for the analysis.

2. Visualization of Commit Data discusses how the data is collected, stored and visualized.

3. Prediction of Commit Data outlines the data and methods that are used to predict change within a repository.

4. Experiments reports the experiments conducted and their results.

5. Conclusion summarizes the results and contributions and proposes future work to build of the thesis.

Chapter 2

Literature Review

In pursue of creating an approach for predicting change within a software repository several areas of research were leveraged. These areas include: Open Source Software (OSS) Repositories, Prediction of Change within Software Repositories and Analysis of Software Repositories. The remainder of this chapter discusses each of these topics in detail and notes the current state of the art as related to each topic.

2.1 Open Source Software

Open Source Software (OSS) generally includes software that provides the ability to access the source code and make modifications to the source code under an open source license. A few examples of open source licenses are: Apache License 2.0[1], GNU General Public License v3.0[2] and MIT License[3]. While certain licenses provide some restrictions on the ability to redistribute the software the main point is that all OSS licenses allow the source code of the software to be freely available. The scope and capability of OSS projects vary greatly and several very popular OSS projects

[1] http://www.apache.org/licenses/LICENSE-2.0
[2] https://www.gnu.org/licenses/gpl-3.0.en.html
[3] https://opensource.org/licenses/MIT

are listed in Table 2.1.

Owner	Project	Description
Mozilla	Firefox[4]	Internet Browser
Linux	Linux Kernel[5]	Operation System Kernel
VideoLAN	VLC[6]	Media Player
PostgreSQL	PostgreSQL[7]	Object-Relational Database Management System
git	git[8]	Version Control System

Table 2.1: Open Source Software Projects

2.1.1 Managing Open Source Software Repositories

The development of large software repositories (whether OSS or not) often make use of Version Control System (VCS). A Version Control System (VCS) helps developers manage repository changes and facilitates collaboration. A VCS will keep a current version of the project and track historical changes (i.e., previous versions) as well. This may be done through keeping a copy of every version of each file in a project or by keeping track of every change made to each file in a project.

2.1.1.1 Version Control System

Apache Subversion (SVN)[9] and git[8] would be two examples of VCS. Git is a Distributed Version Control System (DVCS) and differs greatly from SVN which is a centralized VCS. Git provides the user with a complete local copy of the repository that is available independent of network connectivity. The independence of each user's local repository copy also allows for a application to be developed without a

[4] https://www.mozilla.org/en-US/firefox/desktop/
[5] https://www.kernel.org/
[6] http://www.videolan.org/vlc/index.html
[7] http://www.postgresql.org/
[8] https://git-scm.com/
[9] https://subversion.apache.org/

centralized server. The one main issue with a DVCS is that while decentralization is useful, developers will require some method to collaborate and communicate changes made to the repository. Therefore typically one centralized server is used to maintain communication between all interested parties.

Git provides a simple interface to manage the repository regardless of which site is the central server. Therefore regardless if the project is on GitHub, BitBucket an internal server users can easily interact with the project as long as they know the Git interface. Git in essences is a file storage for the project that keeps track of changes made to the project. A *commit* is a set of changes that a developer makes at a certain time. The developer has full control over what gets committed, when it gets committed.

A branch in a Git repository is a series of commits that are often related. In Figure 2.1, each dot would represent a commit and a set of dots connected by the same colored lines are a branch. Branches can be considered different paths or deviations in the development from each other allowing for different versions of the project to be maintained and developed. The *master* branch is the main branch, represented with black, from which all branches usually stem from and is generally where projects are developed on. On a similar note, a *tag* is a branch that is frozen to allow for future reference. Tags are often uses to mark a significant point in the development history such as a project release. Finally, when two differently branches converge into a single dot then the two branches have been *merged*. A merge indicates that the differences between the two branches are consolidated based on the developer's discretion.

A commit consists of files that have been changed, more specifically a list of *patch* files which each outline the changes made to their corresponding file. The patch file consists of a series of differences between the previous version of the file and this new version of the file. These patch files are key since they contain the actual changes

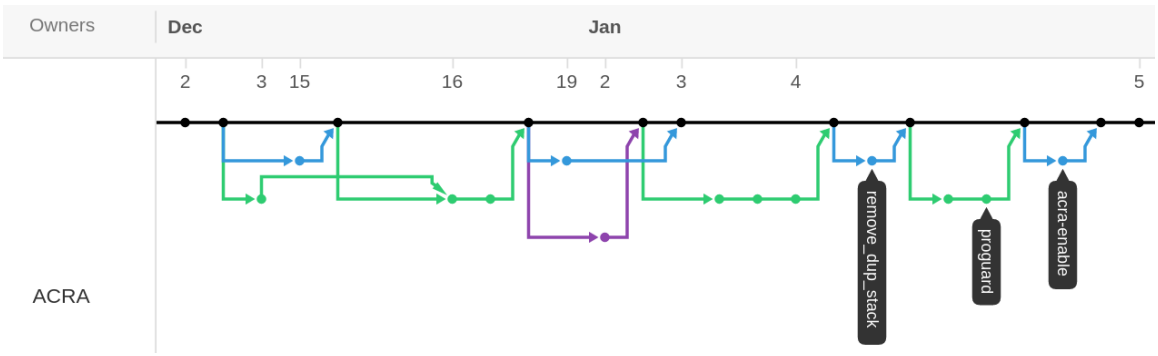


Figure 2.1: Network diagrams

made to the project and thus are the major point of interest.

2.1.1.2 Version Control Management

Git has grown in popularity since it was created and is at the core of several Version Control Management (VCM) sites such as GitHub [10], BitBucket [11] and GitLab [12]. These platforms tend to be fairly supportive of OSS projects through providing their services free of charge. For example, GitHub provides unlimited public repositories completely free. While projects on these sites do not have to be licensed with an open source license they are mostly publicly visible.

GitHub is the most popular of the VCM websites and hosts numerous popular OSS projects including the Linus Kernel[13], Swift[14] and React[15]. GitHub also provides a public Application Programming Interface (API) to allow for data access project repositories (discussed further below). Given the developer popularity of GitHub and the availability of the repository data, GitHub is an ideal choice for data mining repository. All publicly visible GitHub repositories are also publicly accessible

[10] `https://github.com/`
[11] `https://bitbucket.org/`
[12] `https://gitlab.com/`
[13] `https://www.kernel.org/`
[14] `https://swift.org/`
[15] `https://facebook.github.io/react/`

through the API.

2.2 Understanding and Predicting Software Repository Change

The development and maintenance of large scale projects can take years or even decades and involve a huge investment of time and resources. Throughout development of a project developers will make changes to a project. When a developer creates a change to apply to the project it can introduce new functionality or fix bugs. A change may also contain unintentional bugs and require them to be address in the future. Tracing and monitoring changes made by developers can insure that after changes are made the project maintains the correct course and aligns closely with the desired direction of the project. Change predictions are an attempt to potentially know what changes will occur prior to the changes taking place. With the knowledge of what changes will occur the changes can be more informed. For example if a section of the source code is identified as likely to change the necessary resources can be allocated such as a developer who works on that section the most can be tasked with changing it. Similarly, knowing which changes are likely to happen can help developers prioritize their resources according to the importance of each change.

Software development prediction contains numerous areas of study which generally attempt to improve individual projects by focusing on their development and providing feedback and recommendations to developers. Some examples of this work includes: fault prediction [34, 36, 41, 43], mutation score prediction [24], software changes prediction [3, 8, 14, 21, 25, 45]. While there may be a large overlap in the objective for these studies often they will vary in the repository data and prediction method utilized.

2.2.1 Change Analysis

Changes occur within a repository to achieve a specific goal or task. The task can be high-level, such as implement a new features for the program, or lower-level, like fix a syntactical bug. Investigations into how changes are made or used can help provide a better understanding for making a better changes or better use of the changes.

Bieman, Andrews and Yang study the change-proneness of different entities within a software project [5]. In order to provide a deeper understanding visualizations were also used. Koru and Liu studied and describe change-prone classes found within open source projects. Providing further details into characteristics of different changes that are made to a software project throughout development. Similarly Wilkerson attempts to classify different types of changes that occur to a project throughout development. The classification can then be used to identify the impact that a given change will have on other aspects of the project. Snipes, Robinson and Murphy-Hill provide a tool that attempts to locate areas within the source code that have a large amount of changes [42]. These areas could be classified as under development and are likely to be very unstable given the amount of change occurring within them.

2.2.2 Software Development Prediction

Predicting faults and changes within a software repository can allow for the developers or managers create strategies to counter act the negative impact of both faults and changes. Intuitively, finding a fault within a repository earlier can help reduce the cost of repair and number of bugs present in the shipped software version. With a less bug prone software application the end users are less likely to encounter a bug and are able to use the application as intended. The benefits of predicting change are more related to the development process allowing for developers to devise strategies

and effectively implement changes functioning as expected with less faults.

2.2.2.1 Fault Prediction

Fault prediction is a key area of study for software development since the goal is to provide insight into where issues within the project's source code are located. Identifying these areas can be very beneficial by in saving debugging time and efforts. That saved time can be used instead on fixing those issues. Therefore accurate identification of faulty code improves both development efficiency and software product quality. In order to predict these faults, existing approaches used one or more of the following data sources: change metrics [34, 36, 41], code metrics [34, 43], defect history [41], software dependencies [36].

Fault predictions using static and change metrics are studied by Moser, Pedrycz and Succi [34]. The change metrics used outperformed the static metrics in accuracy, and recall. Sisman and Kak alternatively look specifically at change metrics using Information Retrieval (IR) framework to provide the predictions. The prediction framework also uses a time sensitive factor to bias towards more recent changes for predictions. Nagappan and Ball attempted to predict post release project failures for commercial projects. These predictions were done using a software dependency analysis as well as using churn metrics from the project's development. Their method proved to be capable in predicting these failures providing an ability to mitigate these failures from occurring.

2.2.2.2 Change Prediction

The ability to analyze and predict changes within a project could give deep insights into the development of a project. A large amount of research as focused on predictions of changes based on changes [3, 8, 14, 21, 25, 45].

Ying, Murphy, Ng and Chu-Carroll present a method that predicts which parts of the system will change given a set of changes or change propagation [45]. The prediction is done using the project's change history. The results of the prediction method were mixed with some projects recording a stronger precision and recall and others recording a far lower results. Kagdi and Maletic also leverage version history changes to perform software change predictions. The actually analysis applied is two fold, through the dependency analysis of the current version and the change analysis of the version history. The data is collected through Mining Software Repositories (MSR) which is a popular field of study. In a similar work, Hassan and Holt, worked towards predicting change propagation of a given initial change. The main question was to determine given a change to an entity (e.g. function or variable) will propagate to changes in other entities. This work is very related since it tests various methods and leverages presents the best one. Bantelay, Zanjani and Kagdi propose a method that mines the file and method level evolutionary couplings to attempt to predict commits and other interactions within the project [3]. Both methods were used in isolation as well to determine whether the attributes were more helpful when used together. Giger, Pinzger and Gall attempt to build off of previous work in change proneness by providing predictions relating to more refined entities [14]. While typical change analysis will involve the use of syntactic changes. Giger, Pinzger and Gall suggest that extracting and tracking semantic change could prove to be more helpful and accessible for developers for predicting future changes within a project [14]. Chaturvedi, Kapur, Anand and Singh attempt to predict the complexity of code changes to a project [8]. The project's change history is analyzed and the entropy is calculated. The future amount of changes necessary, the complexity of code changes, is then predicted.

2.3 Technologies for Analyzing Open Source Software Repositories

Various technologies are available to help with the analysis of OSS repositories. These technologies include: data mining, visualizations, and machine learning algorithms. Each of these technologies is leveraged in the approach and therefore the previous work is outlined for each technology. The related work primarily focuses on works that are related to software development within a repository environment.

2.3.1 Data Mining

Many data source exist in states that are not convenient or feasible for use without leveraging data mining techniques to transform the data to a more accessible state. These sources of data can vary greatly based on the interests for the individual(s) collecting the data. Examples of data mining techniques have focused both on single source mining and mining of multiple data sources. One application area for data mining is on data collection from software repositories which can be either single or multiple source [11, 20, 22, 25, 31, 48].

Hemmati, Nadi, Baysal, Kononenko, Wang, Holmes and Godfrey take a comprehensive review at the research related to MSR [22]. Several best practices are proposed and areas of future work are identified. Best practices are divided into 4 categories and summarized below.

The first category relates to the data mining which outlines several key points. Firstly, any assumptions or heuristics used in the data collection process require testing to ensure collection is valid. Secondly, the code extraction process is should be chosen based on needs of method as well as available resources. Thirdly, text based data can be expensive to use because of the need to pre-process the data through

methods such as splitting, stemming, normalization and smoothing. Finally, any analysis applied to the data requires verification to ensure the outcome is accurate.

The second category is related to use of the mined data in machine learning algorithms. Firstly, carefully tune parameters and perform sensitivity analyses to improve modeling. Secondly, a more complex objective will likely be less successful than a more simpler objective. Finally a good understanding of the assumptions made by the machine learning algorithm is important to overcoming particular issues within the data set.

The third category outlines best practices related analysis of the data. Firstly, applying a correlation analysis can help determine the validity of the hypotheses prior to a full commitment. Second, use of other characteristics beyond precision and recall may be appropriate. Finally, ensure that research results, data, tools and techniques are shared to community to allow for stronger scientific work in the future

Hassan discusses the value of data mining from software repositories. The possible uses of the data collected can be used towards are assisting developers or managers.

Zimmermann, Weißgerber, Diehl and Zeller collect the change history of a software project to predict changes that should be made in relation to an initial set of changes [48]. The recommendations their tool provides helps point the developer to make changes that are more common within the project. As well their approach can be used to detect which changes may be missed by a developer when making changes to a project.

Maletic and Collard investigated source code changes during a software project's development cycle [31]. The changes are extracted, processed and stored to be more easily analyzed. Canfora, Cerulo and Di Penta propose a method for extracting and refining the changes made throughout the life a project to be used in more effective analyses [7]. The changes made to a project are refined through linking lines of source

code that are related.

A benchmark data set of software project development change history is provided by Dit, Holtzhauer, Poshyvanyk and Kagdi [11]. The data set is processed to provide change request description and tracing, where changes that are requested are able to be traced to where they were implemented within the source code. The data set also provides a corpus of various key aspects of the project including files, classes and methods. The data set is targeted to be used for providing a benchmark for tools attempting to improve software maintenance tasks.

2.3.2 Visualization

Visualization techniques are often used of changing the way the data is represented to help with comprehension and ease of use [5, 13, 26]. Visualizations is a common technique used in combination with data mining. It can often used to represent data to be more accessible to appealing for use. Rather than view large amounts of complex data a visualization can restrict the amount of information shown to prevent the user from being overwhelmed. Alternatively, a well designed visual representation of the data can retain underlying information and represent it in a way that is more convenient. Visualizations are widely used throughout software engineering research to represent software evolution [5, 9, 13, 17, 26, 38] and developer interactions [10, 15, 17, 38, 39].

Some of the visualizations attempt to focus on a particular aspect of a dataset. Gall and Lanza discuss uses of project traits including the source code changes, release information and quality metrics to provide the necessary data for powerful visualizations [13]. Similarly, Collberg, Kobourov, Nagra, Pitts and Wampler use CSV version control systems to visualize software evolution [9]. A visualization is produced which provides a temporal element for development data. Another approach studied visual-

ization a change-proneness within the software project [5]. Lanza, Ducasse, Gall and Pinzger present a high level visualization tool for object-oriented projects [30]. Ogawa and Ma create a story view visualization for software projects with an summary of changes for each commit [39]. The visualization can have issues with large amount of information available.

Ogawa and Ma provide a expansive visualization which includes developer and source code interactions [38]. Gonzalez, Theron, Telea and Garcia visualize a combination of software project metric and structural changes. Four designs are proposed to provide unique and complementary views.

2.3.3 Machine Learning

Machine learning is a complex method for software algorithms to attempt to determine patterns within the data. One such problem example would be an algorithm to detect certain people within an image. For an individual such a task may seem trivial however for a software system to detect it is far more difficult. Algorithms that can determine patterns and mimic them from abstract set of data is useful when such patterns are extremely complex. There are numerous algorithms which apply machine learning approaches. Each approach has both advantages or disadvantages. Some examples of machine learning algorithms are Support Vector Machine (SVM)[16], Random Forest (RF)[17] and Artificial Neural Networks (ANN)[18]. The three provided examples are also commonly used for data mining [1, 4, 18, 23, 24, 46]. Bhattacharyya, Jha, Tharakunnel and Westland provide a detailed description of Random Forest (RF) and Support Vector Machine (SVM) [4].

[16] http://www.support-vector-machines.org/
[17] http://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm
[18] https://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.html

2.3.3.1 Support Vector Machines

A SVM is used to predict what type of change will occur based on a set of features provided. A feature is a data extracted from the project represented as a floating point number. In order to be useful a feature must in some way characterize the the category that it is assigned to. The feature must also not rely on the category that it belongs to in order to be calculated. For example, given a category of the method change within the next 5 commits or not, then the features must not rely on knowledge of future changes to the project. If the features fail to effectively characterize the category they are assigned to then the SVM may have poor predictions. It is also necessary for the features to independent of each other to not negatively affect the categorization.

SVM requires all feature data be encoded as floating point numbers. For any numerical data the conversion to floating point is trivial. However, for more complex data the conversion is a little more difficult. Categorical data can be mapped into a unique vector entry per category. For example, if a feature can be 1 of 3 options: 0, 1 or 2 then it can be converted into three entries in the feature vector. Encoding the value 2 the sub-vector of the feature set would be {0, 0, 1} where 1 indicates a field that feature is present in the data for this vector, and 0 indicates the feature is not present. Data that is in the form of a string can be converted to a floating point number by assigning a unique number for each string (similar to hashing). The one downside to this method is that the numbers corresponding to each string maintain no numerical properties. In essence the data becomes categorical, such that if *bob* is mapped to 1 and *sally* is mapped to 2 there is no relationship between 1 and 2. Ideally, this data would then be further converted using the previously described method however if the set of possible strings is large then it may be unreasonable to convert it. For example, if there are 100 possible strings then that would add 100 new entries to a single vector.

The categorization is used for the prediction, where each value of the category relates to a unique prediction type. For example, a simple binary categorization could simply 1 or 0 where 1 predicts the event will occur and 0 predicts that the event will not occur. In essence an SVM is tasked with separating a dataset into two different categories given a sample set of data that has already been categorized into two subsets. Given the categorization of the sample dataset the SVM model is trained to allow for categorization of new data. The categorization of any new vectors (that were not used for training) is called a prediction and is made by the SVM model created through the training. More specifically, the sample dataset is a dataset extracted from the target dataset. The sample dataset is then categorized based on the predetermined criteria (the prediction goal). This dataset along with the categorization for each vector in the dataset is the training dataset, and is then used to *train* the SVM model. Once the model has been trained, the SVM model is ready to be used for making classification predictions. The data for each feature can be extracted from the new dataset, allowing for the model to classify each new vector. Given that the SVM model is accurate and reliable the results can then be used towards making predictions about the dataset. For example if the classification is that of predicting change to occur within the next six commits the developer may wish to be careful with the use of the method or assess the method's quality and determine if any issues within the method need to be addressed.

A lower prediction score often relates to the data from the feature set poorly characterizing the categories. Similarly a warning will be given if the dataset is inseparate. In this case, the dataset for each category may be too similar and cannot be properly split into the two category subsets. In both cases a change to the feature set may help, whether that is a decrease or increase of features in the set. Some features are detrimental to the model, especially two features related to one another.

More details about the specific features used will be given a little later on. Features are descriptive aspects of the dataset that are classified into the predetermined categories. Since these features relate directly to the category understanding of the classification critical and can help determine which features should be used. For example for a classification of whether a change will occur within the next few commits, a useful feature may be the frequency by which a method changes within the project. Picking a descriptive feature set is paramount to providing a strong prediction of future data.

SVM has been widely used for making predictions for various aspects including predicting battery charge state [2], pharmaceutical data [6], software faults [12, 16, 29, 32, 33, 37], bug localization [35, 37], software mutation testing score [24], financial stocks [28], credit score [23], credit card fraud [4], solar power output [47].

Malhotra reviews numerous machine learning techniques, including SVM and RF, used by various studies [32]. The results of which outline where each approaches succeed and falls short. When using a machine learning algorithm it is imperative to use a suitable algorithm for current situation. Kim, Whitehead and Zhang outline a approach that uses a SVM to predict changes that will occur within the project [29]. By identifying these changes the a project developer can potentially locate a bug within a change and fix it prior to being reported. Erturk and Sezer compare the performance of their proposed method, an Adaptive Neuro Fuzzy Inference System, to that of an SVM for predicting software faults [12]. The models are trained using project metrics as well as the project's historical fault data. Zeng and Qiao use a SVM to provide short-term predict solar power output [47]. The SVM model outperformed both an autoregressive and a neural network model. Antón, Nieto, Viejo and Vilán propose a method for predicting the state of charge of a battery using SVM model [2]. Neuhaus, Zimmermann, Holler and Zeller mine vulnerability databases and version

archives to determine components within the software that were vulnerable [37]. A SVM was then used to predict other component that were also vulnerable. Several feature selection techniques have been assessed by Shivaji, Whitehead, Akella and Kim for bug prediction methods [40]. Features which are less useful to the prediction are removed to reduce the set to only the essential features. Kim investigates the possible use of SVM as a prediction model for financial forecasting. The model was used to predict whether the stock price would go up or down for the next day. Bhattacharyya, Jha, Tharakunnel and Westland uses RF, SVM, logistic regression to detect credit card fraud [4]. Both RF and SVM are able to predict a large number of fraudulent credit card transactions.

2.3.3.2 Random Forests

RF are a popular machine learning algorithm and is used in numerous areas including predictions for software fault [19, 32, 33], software development effort [33], credit card fraud [4], database indexing [46], malware detection [1].

Malhotra provides an extensive review of studies involving machine learning to predict software faults [32]. The results showed that RF tended to preform better than other machine learning algorithms studied. Moeyersoms, Fortuny, Dejaeger, Baesens and Martens made use of RF and SVM as well as a few other data mining approaches to predict software faults and effort estimation [33]. The data mining techniques are used as part of another model, ALPA rule extraction, to improve the predictions and increase traceability. Guo, Ma, Cukic and Singh attempt use RF to predict the fault proneness of modules within a project [19]. The RF prediction results for the five sample projects prove more accurate to that of a logistic regression. Yu, Yaun and Liu attempt to use RF to determine a more effective database indexing for video data [46]. The database index are used to provide faster searching of the

database for action detection.

RFs are commonly used to make predictions on data that has been mined from some source [1], [18], [46]. A RF leverages numerous decision trees to provide attempt to improve prediction capabilities. Therefore to fully understand a RF first an understanding of decision trees is necessary. A decision tree is a technique which will create a tree based on a data set that has been classified. Once the decision tree model is created it can be used to predict or categorize data that has not yet to be classified. In the tree model the leafs will be categorizations where as the connections between inner nodes are the decisions by which the categorizations are made.

One issue with decisions trees and more generally machine learning techniques in general is imbalanced data sets for training the model [27]. The data set used rarely provided even sample sizes of each set therefore without taking necessary pro-cautions the algorithm will bias the results. In the worse case the model will classify any input data as the larger data classification.

In case of imbalanced datasets there are several methods to help provide stronger predictions [27]. The most obvious and easiest to attempt would be to sample more data. However if the dataset in general follows this trend then some more advanced techniques can used to improve the model.

The first method would be to *undersample* larger category this will even out both of the categories. This will remove some of the input values within the dataset to reduce the set size. However if there are very few samples of the smaller category the performance will suffer as well. A second method of *oversampling* is useful in the case were the data samples are small. The input data from the smaller category is selected to be duplicated in the set to increase the size of the set. This helpful since it will increase the size of the dataset but could lead to bias based on the data selected from the smaller dataset. The selection method for which input vectors to

over or under sample can be based off on the data's statistical distribution or made by random choice. Another advantage of these over and under sampling is that they can also be used together to in the case of a large disparity between the category's set size.

Another feature of RF which helps provide more reliable predictions is *Bootstrap Aggregation* [4]. Similar to normal sampling methods it will start with the initial dataset. However, rather than using the dataset as is the dataset will be uniformly sampled n times and repeated m times to create m datasets of n values. These newly created datasets will then be used to train m models. Finally, when attempting to categorize a new input data it will be given to every model and the prediction result will be aggregated to provide a more accurate results. For some other machine learning methods, such as SVM, this method will improve the results and help with imbalanced datasets.

A RF is a collection of decisions trees trained on random samples of the initial dataset. So the RF will take an input dataset and then train m decisions trees using m randomly sampled sub-datasets of the initial dataset. This helps improve the model created and makes RFs far easier to use. As well RFs have a feature that determines the importance of each feature is assessed during the training of the model [4]. The importance outlines the quality of each feature in providing the prediction [44]. Therefore in order to properly understand the feature importance the accuracy, precision and recall of the model should be determined by running a test dataset to determine the quality of the model.

Chapter 3

Visualizating Commit Data

The proposed research focuses on predicting changes within a software project. This is accomplished through mining of software data, analysis of collected data, candidate feature analysis. After the raw data has been collected a further analysis is used to extract key features. As part of this analysis, custom visualizations are used to help to provide insights into the data set.

3.1 Data Mining

Before we can predict changes within a repository (see Chapter 4), we must first collect data. The data collection is targeted towards Open Source Software (OSS) projects that are predominately written in Java[1] and hosted on GitHub. Some of the these repositories, especially larger ones may be multilingual and include other programming languages for purposes such as a database schema outline. The purposed data collection approach is not language specific, however in order to simplify the implementation, the approach was restricted to only collect Java repositories. Generalizing our implementation for other languages may require a redesign especially with

[1]A project would be predominately written in Java if it has over 75% of the source code in Java.

respect to collecting the prediction model features. The data collection process simply data mines the complete development history of the project through the commits stored in GitHub. The commit data includes:

1. Developers related.

2. Source files.

3. Changes made in the commit.

4. Project release information in the form of tags.

The raw data is kept unprocessed and stored directly in a relational database (MySQL) that allows the data to be used and manipulated without requiring further access to GitHub. The collection process can be lengthy and depends largely on the size of the repository. For example, smaller repositories such as acra or tempto can take a matter of minutes to collect. However, repositories that are far larger like deeplearning and storm can take several hours to complete. In the case of an incomplete or interrupted collection, the process can be resumed to collect the remaining data at a later time. Similarly, a repository that was previously collected can be mined a second time to collect any new commits made since the previous collection. These maintenance collections will often be much smaller and require less time.

Our collection method for mining data from GitHub repositories utilized GitHub's web Application Programming Interface (API)[2]. The GitHub web API allows automatic access to the complete data set of publicly available repositories. The collection process requires both the developer and repository names in order to work. To actually collect the data from GitHub a Ruby script, built around a Ruby library, *github_api*[3], was used as a convenient wrapper for GitHub's web API. The script sys-

[2] `https://developer.github.com/v3/`
[3] `https://github.com/piotrmurach/github`

tematically collects all of the desired data related from a given developer's GitHub repository.

Some unnecessary aspects of a GitHub repository's dataset are not collected but could be with a minor extension to the collection script. Aspects not collected include:

1. The **issues** data outlines the problems reported in the project by users or developers of that project. GitHub allows for issues to be optional and thus some projects do not offer issue reporting through GitHub.

2. Branches are also directly related to the repository and they are essentially different workspaces for the developers. They allow for development of different versions (such as a development version compared to a stable version). For simplicities sake, the approach assumes that the main branch (the master branch) is the development branch and the target of the analysis. Therefore the branches are ignored at least for the scope of this work and could be included in future work. Of course other branches could be analyzed however the perspective of the other branches typically originates from the master branch.

3. Forks are more peripheral than branches but follow a similar scheme. A repository fork can be created another develop who does not own the current repository. The fork in essence is a branch that is owned and controlled by the developer who created the fork. A fork is therefore outside of the scope for this approach since it outside both the main repository workspace.

4. Pull requests are created by developers to request the owner or collaborators accept changes to the project. This mechanism facilitates developers who are not affiliated with the repository owners to change the project. The owner can accept, reject or request further changes of the pull request to manage the developer's contribution.

A similar subset of data not collected or used for this approach is any forks of the repository. In GitHub a fork is an externally created branch of the project. The major differences between a fork and a branch are that a fork is owned by another developer and a fork is in fact a project onto itself. This allows for a developer who are not contributors to make a copy of the project and work on it without affecting the original. Forks typically denote a deviation from the original project that is unlikely to be reconciled. Finally, pull requests facilitate external developers making small changes which tend to be fixes to problems found or desired feature implementation. The owner of the original repository can then decide to integrate the changes made the original repository.

3.2 Storage

As mentioned above the raw commit data is stored in a MySQL relational database which leverages the Structured Query Language (SQL). There are three databases used for the collection and the analysis of commit data. The first stores the raw mined data, whereas the second stores the analyzed data in a more convenient layout to be used later. Finally, the third database stores the same data as the second however uses a different relational database implementation because of some limitations within MySQL. This third database uses PostgreSQL, which has a more advanced set of features than MySQL and is simply a clone of the data stored in the second database. The specific limitations that were encountered will be discussed more fully later in this section.

The first database, *github_data*, stores the semi-raw data collected from GitHub's API. This database contains 8 tables which store various aspects about the projects considered potentially important for the analysis later on. The tables of primary

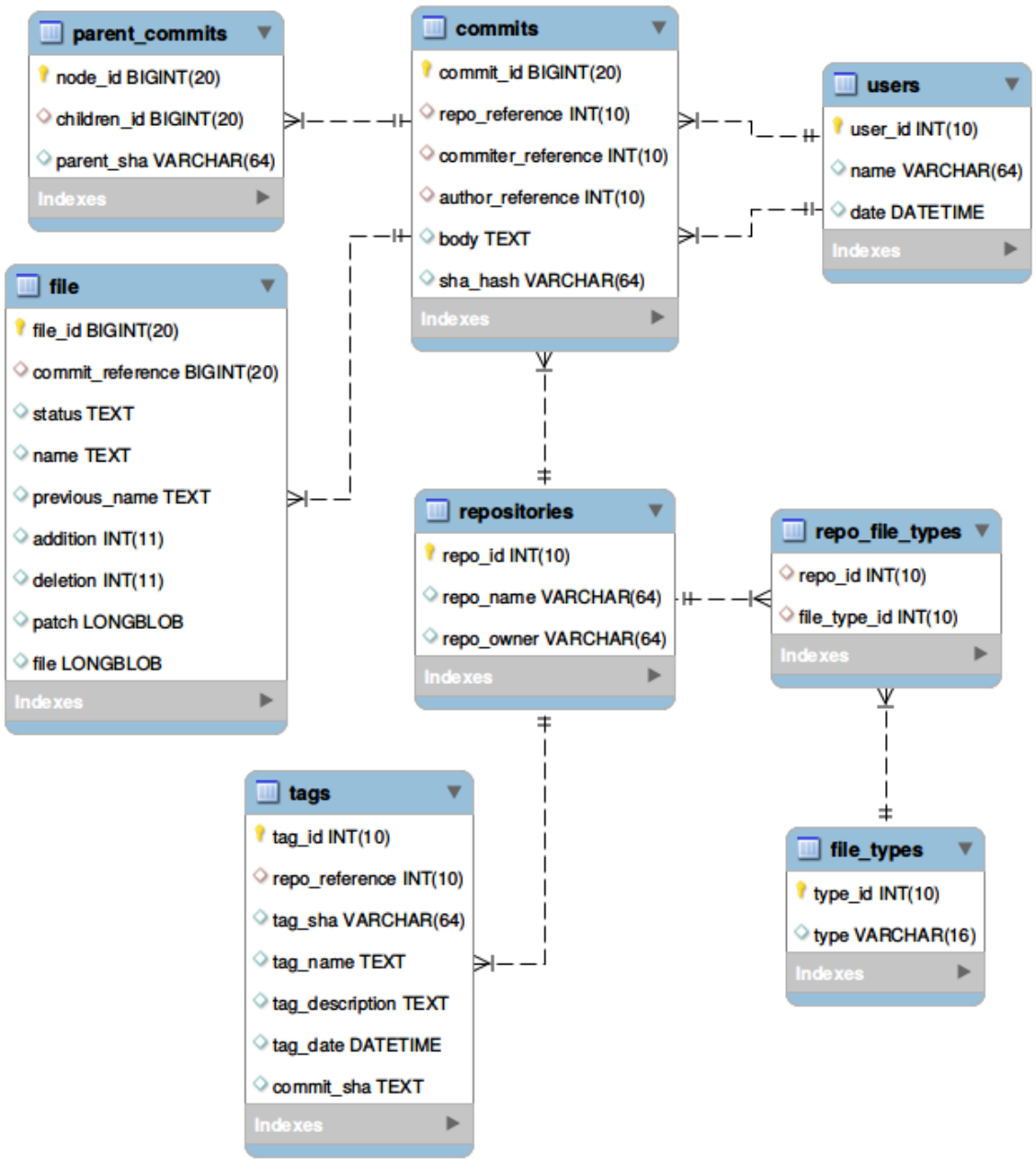


Figure 3.1: GitHub Data Schema

concern are *repositories*, *commits*, *users*, *files* and *tags* tables. The complete schema is outlined in Figure 3.1. Other aspects are available from the API and if needed the database could be extended to store more elements as necessary. In some cases, data from the API is not available for one reason or another (usually inaccessible

files or such) these are simply removed or a note is made of them depending on their importance. For example, non Java source files that are missing are not essential and if inaccessible are ignored. If a Java file is inaccessible, a note is made as this is a greater concern. These files can be retrieved if enough information is available (previous version and corresponding patch file). In the case that insufficient information is available, the analysis can still be applied but will likely adversely effect the result.

After storing the data in the *github_data* database, the analysis process is done. The *parsing* script is run next and discussed further in the Section 3.3. The results are then stored in the *project_stats* database that is very similar in layout to the first database except some extra tables have been added and a few data items have been removed. Mostly the storage expansions are designed to hold change information calculated from the analysis of the data. The complete schema is outlined in Figure 3.2.

The third and final database uses PostgreSQL because of limitations within the MySQL implementation. The calculation of the candidate features, discussed in further detail in Section 5.1, required a more versatile partitioning function and the ability to perform multiple inner queries. The first of which is more difficult to implement and the second is not available at all MySQL. Therefore the data was transferred over to PostgreSQL, using simple program called *pgloader*[4]. Only one difficulty was encountered during the transferring process. One of the tables in the MySQL database was called *user*, however in PostgreSQL, this is a reserved table name and therefore the table cannot be interacted with properly. The work around was to simply rename the table in MySQL prior to transferring to avoid any issues with the database. After transferring the data over to PostgreSQL, the data change predictions are ready to be preformed.

[4] `http://pgloader.io/`

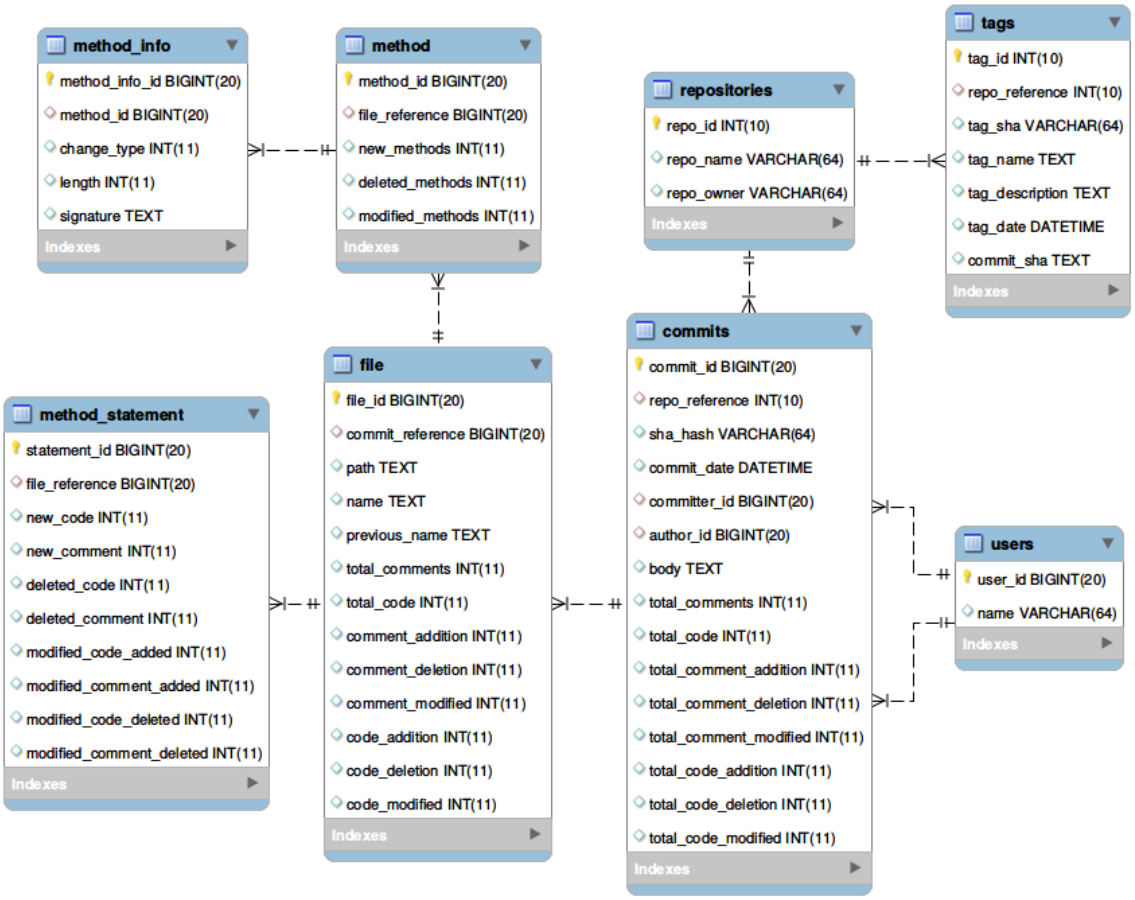


Figure 3.2: Project Stats Schema

3.3 Processing

The raw commit data collected from GitHub is stored and undergoes an analysis to extract more refined details. The process first requires the changes from a commit, the patches, to be merged into their corresponding full file. A patch is simply a stub file which summarizes the changes that occurred within a source file. Once the patch is merged with the raw source file, a full file is formed that contains every change as well as the source code that did not change. Within a patch file and a full source file, three different types of changes are present; additions, deletions and no change. These are represented as a plus sign, minus sign and space respective. An example

is outlined of each of these changes in Figures 3.3, 3.4, 3.5, 3.6. The coloring used within these images is purely for visual effect and not present in raw patch files.

```
+    private static <E, T extends Comparable<T> > Integer binarySearch
+        (ArrayList<HashMap<E, T > > patterns, T target, E attribute,
+        int start, int end) {
+
+        if(start > end) return null;
+
+        // One element left lets check that element
+        if(start == end) {
+            if(patterns.get(start).get(attribute).compareTo(target) == 0) {
+                // The last value is the one we are looking for
+                return start;
+            }
+            return null; // The value is not here.
+        }
+
+        int middle = (start + end) / 2;
+        int result = patterns.get(middle).get(attribute).compareTo(target);
+        if (result > 0)
+            return binarySearch(patterns, target, attribute, middle+1, end);
+        else if(result < 0)
+            return binarySearch(patterns, target, attribute, start, middle-1);
+        return middle;
+    }
```

Figure 3.3: Newly added method

The process of reconstructing a full file using a patch file requires modifying the original source file. Since the source file used is the product of the patch file, the patch must be applied in reverse. Therefore lines with a plus sign, additions, are assumed to be within the file and lines with minus signs, deletions, are assumed to not be present within the file. The lines previous removed from the source file are added back to their original location with a minus sign to preserve the original meaning of the line. The lines that were added to the source file are perpended with a plus sign to identify that the line is an addition.

This full source file is then analyzed to extract each method to identify the type

```
-    private static <E, T extends Comparable<T> > Integer binarySearch
-        (ArrayList<HashMap<E, T > > patterns, T target, E attribute,
-        int start, int end) {
-
-        if(start > end) return null;
-
-        if(start == end) {
-            if(patterns.get(start).get(attribute).compareTo(target) == 0) {
-                // Value Found
-                return start;
-            }
-            return null; // The value is not
-        }
-
-        int mid = (start + end) / 2;
-        int result = patterns.get(mid).get(attribute).compareTo(target);
-        if (result > 0)
-            return binarySearch(patterns, target, attribute, mid+1, end);
-        else if(result < 0)
-            return binarySearch(patterns, target, attribute, start, mid-1);
-        return mid;
-    }
```

Figure 3.4: Removed method

of change as well as other method metrics. The type of change that occurred to a method is identified as one of four possible changes. First, a method may be completely new and is thus classified as a new method shown in Figure 3.3. The second method closely, related to the first would be an entirely removed method that is classified as a deleted method shown in Figure 3.4. The third classification that is more difficult method to identify is a modified method. Simply a modified method is one that contains at least two of following three change types; added, removed or unchanged lines. An example of a method that contains all three change types is shown in Figure 3.5. In the event that a method consists entirely of additions and deletions then the method is classified as both a new method and deleted method. A deleted and added classification is used over a modified classification because if

```
    private static <E, T extends Comparable<T> > Integer binarySearch
        (ArrayList<HashMap<E, T > > patterns, T target, E attribute,
        int start, int end) {

        if(start > end) return null;

        if(start == end) {
            if(patterns.get(start).get(attribute).compareTo(target) == 0) {
                // Value Found
                return start;
            }
            return null; // The value is not
        }

-       int middle = (start + end) / 2;
-       int result = patterns.get(middle).get(attribute).compareTo(target);
+       int mid = (start + end) / 2;
+       int result = patterns.get(mid).get(attribute).compareTo(target);
        if (result > 0)
-           return binarySearch(patterns, target, attribute, middle+1, end);
+           return binarySearch(patterns, target, attribute, mid+1, end);
        else if(result < 0)
-           return binarySearch(patterns, target, attribute, start, middle-1);
-       return middle;
+           return binarySearch(patterns, target, attribute, start, mid-1);
+       return mid;
    }
```

Figure 3.5: Mixed changed method

all lines are deleted and re-added then the method is change far more drastic than a simple modification. The final change type is that of no change, where the method does not contain any changes and is shown in Figure 3.6.

For each commit, this information is stored to allow for easier access and save time since the analysis of larger datasets can be time intensive. In order to maintain the integrity of the initial dataset, this information is stored in a new database. There are several other features available in the data set from the extraction process beyond the ones outlined here in detail. A few of those features include: the commit author, the commit message and the method length per commit. This data is stored in the

```
    private static <E, T extends Comparable<T> > Integer binarySearch
        (ArrayList<HashMap<E, T > > patterns, T target, E attribute,
        int start, int end) {

        if(start > end) return null;

        // One element left lets check that element
        if(start == end) {
            if(patterns.get(start).get(attribute).compareTo(target) == 0) {
                // The last value is the one we are looking for
                return start;
            }
            return null; // The value is not here.
        }

        int middle = (start + end) / 2;
        int result = patterns.get(middle).get(attribute).compareTo(target);
        if (result > 0)
            return binarySearch(patterns, target, attribute, middle+1, end);
        else if(result < 0)
            return binarySearch(patterns, target, attribute, start, middle-1);
        return middle;
    }
```

Figure 3.6: Unchanged method

database to help create the prediction model later on.

3.4 Visualization

3.4.1 Line Change

The key features are extracted from the data set after performing the collection and analysis. Visualizations were used in order to to better understand resulting data. The first visualization in Figure 3.7 simply showed the changes recorded on a per line basis. These changes were divided into several closely related subcategories of additions, deletions and modifications. Additions identify changes that are new and do not have a corresponding set of deleted code. Similarly, deletions refer to

changes that remove lines of code without a corre sponding set of additions. Finally modifications are a set of changes which contain a set of additions and deletions that are related.

The relationship between two sets of additions and deletions is determined through the Levenshtein Distance (LD) formula[5] shown in Equation 3.1. The LD calculation will determine the edit distance between two strings, where edit distance is defined as the number of characters difference between two different strings. For example, the LD for *happy* and *mapper* would be 3, since h would be changed to m, y to e and r would be added at the end. Normalization is used to allow for more general use of LD for comparing strings of different length. To calculate Normalized Levenshtein Distance (NLD) the LD would be divided by the larger of the two strings sizes shown in Equation 3.1.

$$NLD(a_i, d_j) = \frac{LD(a_i, d_j)}{\max(|a_i|, |d_j|)} \tag{3.1}$$

Line modifications are assumed to only take place in a series of line changes that involved both additions and deletions shown in Figure 3.5. In this example, 3 line modifications take place each containing 1 addition and 1 deletion. A line modification can also have an $|a|$ to $|d|$ relationship. That is to say more generally, $|a|$ lines of addition may relate to $|d|$ lines of deletion where both $|a|, |d| > 0$. In order to determine whether two lines are closely related enough, a threshold Δ_m is defined. As outlined in Equation 3.2, when the NLD is below the threshold Δ_m then the two lines are related.

$$m(a_i, d_j) = NLD(a_i, d_j) < \Delta_m \tag{3.2}$$

[5] `http://www.levenshtein.net/`

Normalizing the LD calculation accounts for the differences in line sizes when being compared. With shorter lines, the change of a variable name could change a large portion. Therefore with smaller lines likely modifications result in a dramatically higher distance between lines. Likewise, longer lines can contain more text modifications and still result in a low score because of the length of the line. This resulted in the creation of the a threshold α to separate small and large line changes. The Equation 3.2 is updated accordingly shown in Equation 3.3.

$$m(a_i, d_j) = \begin{cases} NLD(a_i, d_j) < \Delta_s & \text{if } \max(|a_i|, |d_j|) < \alpha \\ NLD(a_i, d_j) < \Delta_l & \text{otherwise} \end{cases} \tag{3.3}$$

Only lines that are part of the same block of additions and deletions are selected for the similarity check to determine whether they can be classified as a modification. As noted before, line modifications will consist of one to many addition lines mapped to one to many lines of deletion. Therefore a modification is more easily referred to as a modification set. For addition lines that do not meet the threshold of similarity with all deletion line in the change block are classified as additions. Similarly, deletion lines that fail to meet the similarity threshold for all addition lines will be classified as deletions. A block of changes will therefore contain a set of added and deleted lines, some of which may be related.

While creating the parsing method, both code and comments were considered separate entities. However each was analyzed with the same method. Therefore for the visualization below, the changes are separated into source code and comments. The comments were never used towards the prediction method presented later on in Chapter 4 and will therefore not be covered as deeply. The comments however are available for use and could be used to extend the approach.

The visualizations are interactive providing a more rich experience when used. The

visualizations are available online with some example repositories[6]. The interactions include:

1. Supplemental information on hover over.
2. Adjustable data range through either set windows or manually adjusted windows.
3. Customizable legend that allows for data sets to be set to hidden or shown.

A snap shot is presented to outline and summarize their overall capabilities. The line change data is visualized in Figure 3.7. The number of changes lines of source code added, deleted and modifications are all shown aggregated per month. A per commit view is available but is very cluttered because of the excess of data with in a project. The bottom half of the visualization shows the sum of changes up till the given point. Tags for the project are shown at the bottom of the graph to provide some context of the release cycle. Tags often mark points of significance within the project and therefore can be thought as road signs. The visualization also provides some options to refine or generalize the view. For all of the views, the user is allowed to select the project, package path, and the committers as desired. Specifically for the line level graph, a further option is provided to condense the data based on a monthly, weekly and commit summary. The commit message and a link to the commit on GitHub are provided when viewing either the commit view, method level or statement level. This information allows for a direct link to the project and can be a handy tool for referring back to the software repository.

[6] `http://sqrlab.ca/gitview/`

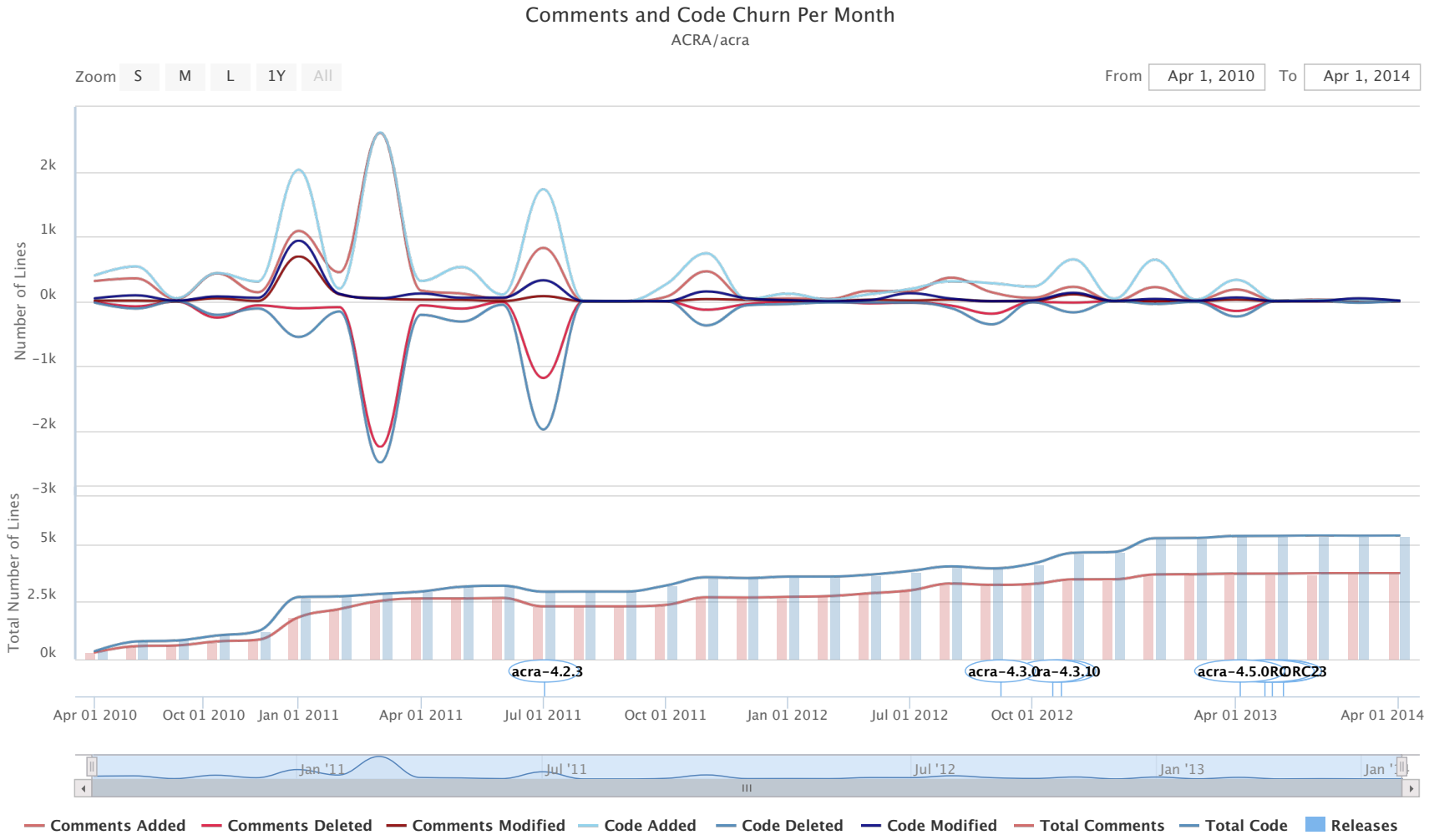


Figure 3.7: Line Change Visualization for acra

3.4.2 Method Change

The visualization of line changes was very noisy and proved difficult to use. Instead of viewing every line of change separately, the changes are grouped together based on the method from which they originate from. Similar classifications are used for method changes however their definitions vary slightly and are outlined in more detail in Section 3.3. There are three types of method level changes that can occur:

1. a method is classified as newly added when that method had not existed in the previous version, consisting only of additions.

2. a deleted method implies that the method is completely removed from the current version, consisting only of deletions.

3. a method is classified as modified if it contains two or more types of changes, either additions, deletions or no change.

The method level change visualization, shown in Figure 3.8, presents the amount of method changes that occurs in the project development over time for acra. The low level changes details are ignored in this view, instead the focus is placed on that of the three types of changes. The visualization for the method level uses a bar graph since it provided a more clear picture of the relationship between commits. Compared to the first visualization which implied that a relationship between different commits of the same type of changes. The contrast in granularity between each type of change and each commit is also more clear and defines the visualization. The amount of change occurring over time is clearly visible and the amount of data available is not as overwhelming as the line change visualization.

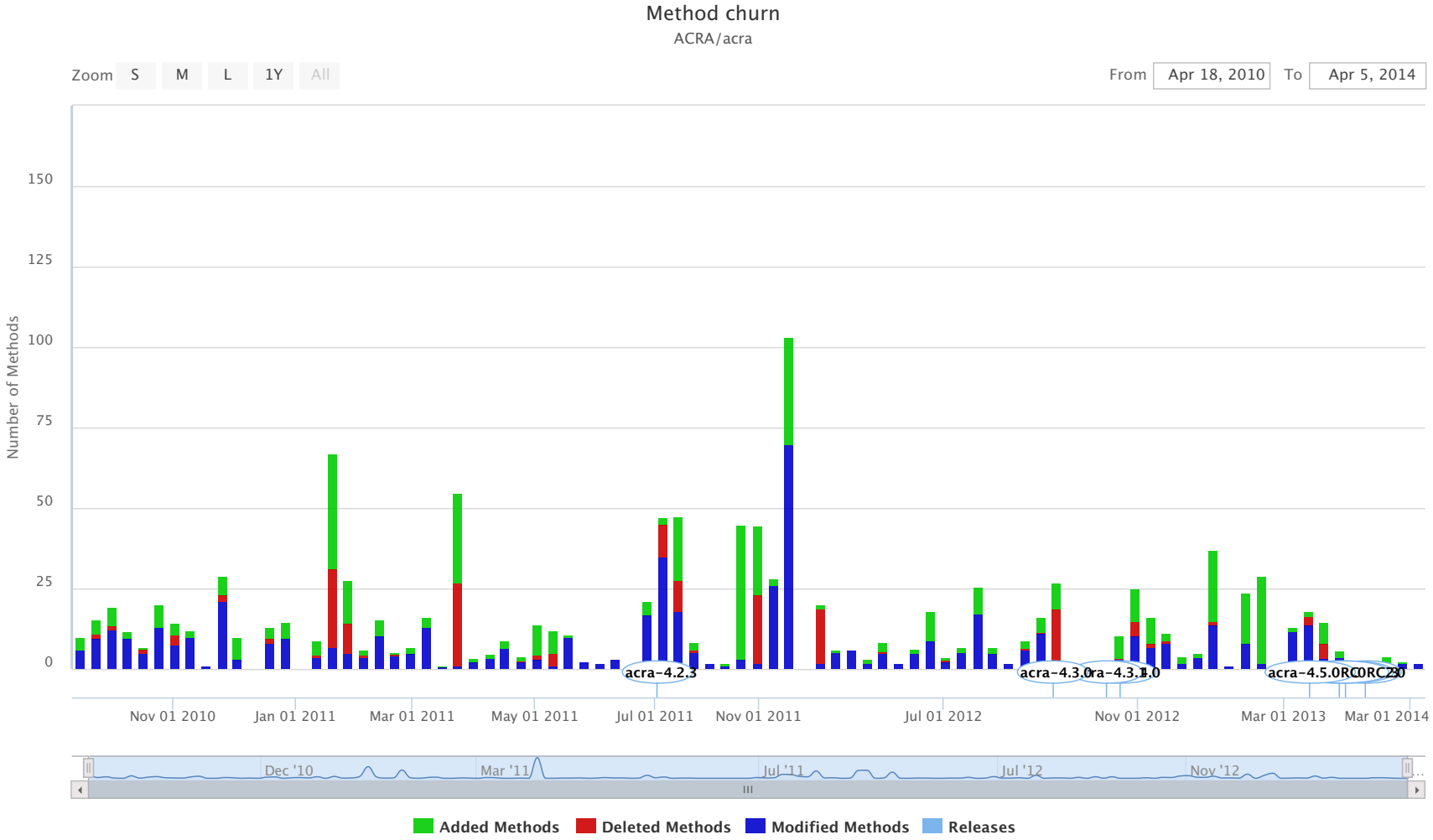


Figure 3.8: Method Change Visualization for acra

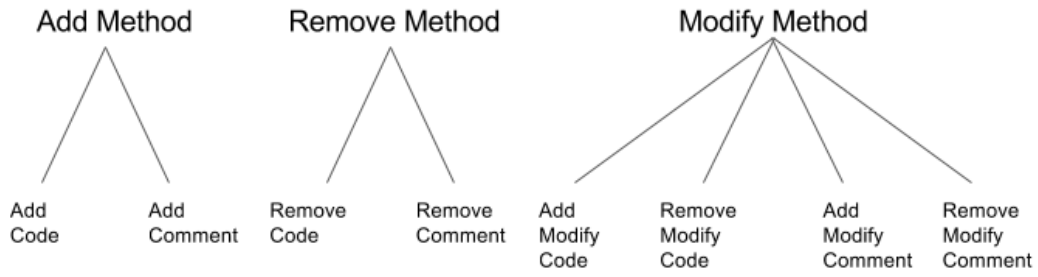


Figure 3.9: Method Statement Categories

3.4.3 Method Statement Change

The method statement level visualization is a more granular view of the method level view. By building on top of the method level view, the method statement level view provides more details similar to that of the line level view. The classifications are kept from the method level view for changes but are broken down into line changes made to comments and code. Therefore both methods consist of two parts, the comments and the code that are added, deleted and modified based on the previous classification and are divided into new categories. Added and deleted methods are divided into two new categories each; added code, added comment, deleted code and deleted comment respectively. The list of classifications are outlined in Figure 3.9, where the second tier is the new classifications of statement changes. In Figure 3.10, the added and removed method data is shown. The modifications are disabled in the example visualization to simplify the view.

In Figure 3.11 the modifications for method statement changes are shown for acra. Modifications are divided into four categories instead of two. The first two categories relate to modification of comments, and the second two relate to modifications of code. The comments changes are classified as either modified added or deleted comments. Likewise for source code modifications, the classification is either modified added or deleted code. Each line classified under modification is based on the change type of

the method. Therefore if a line of source code is part of a method that is modified then it will fall into one of the four modification classifications. For example in Figure 3.5, there would be 5 lines classified as a modified deleted code, 5 lines of modified added code and 0 lines of modified added or deleted comments.

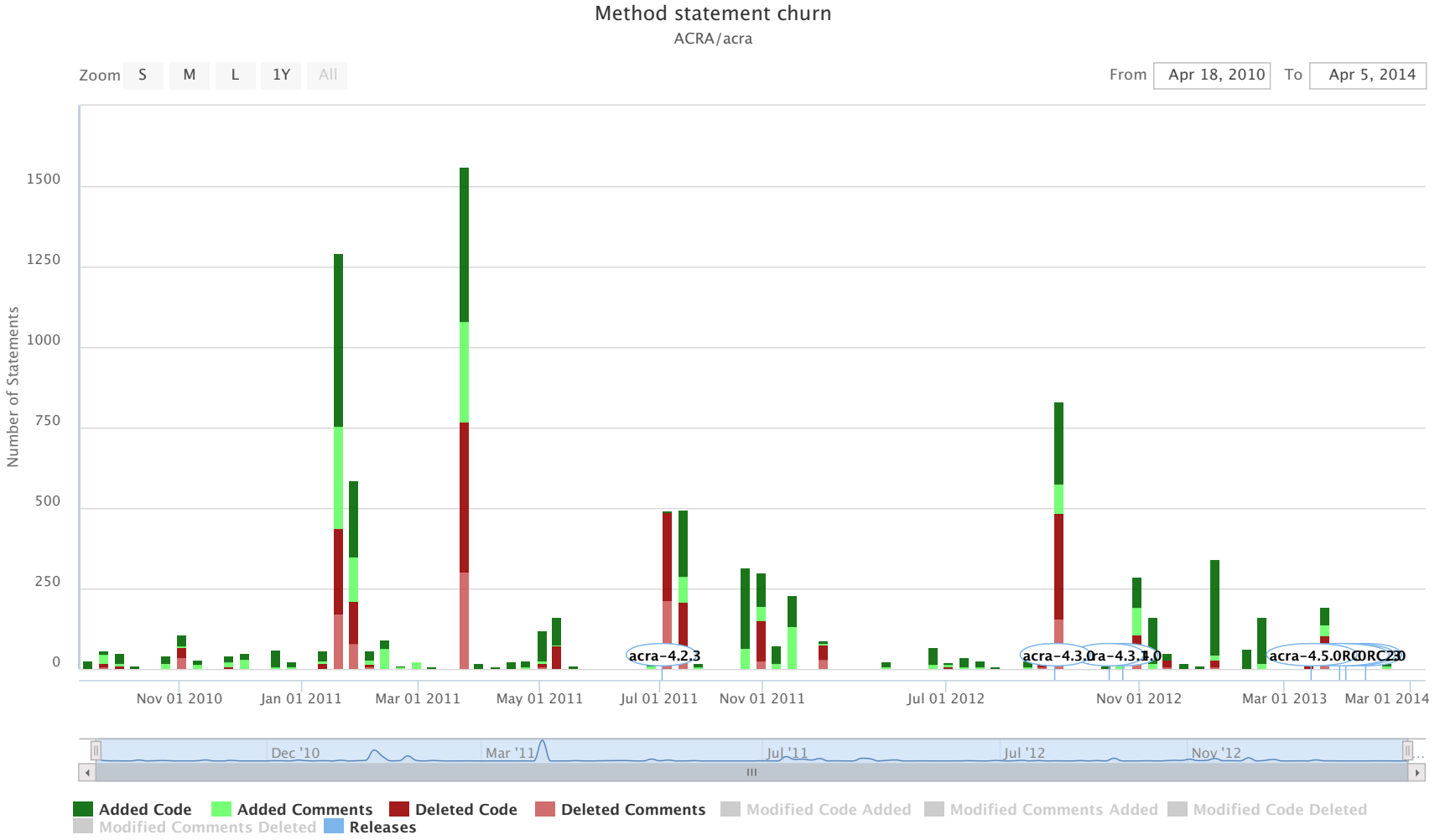


Figure 3.10: Method Statement Added & Deleted Visualization for acra

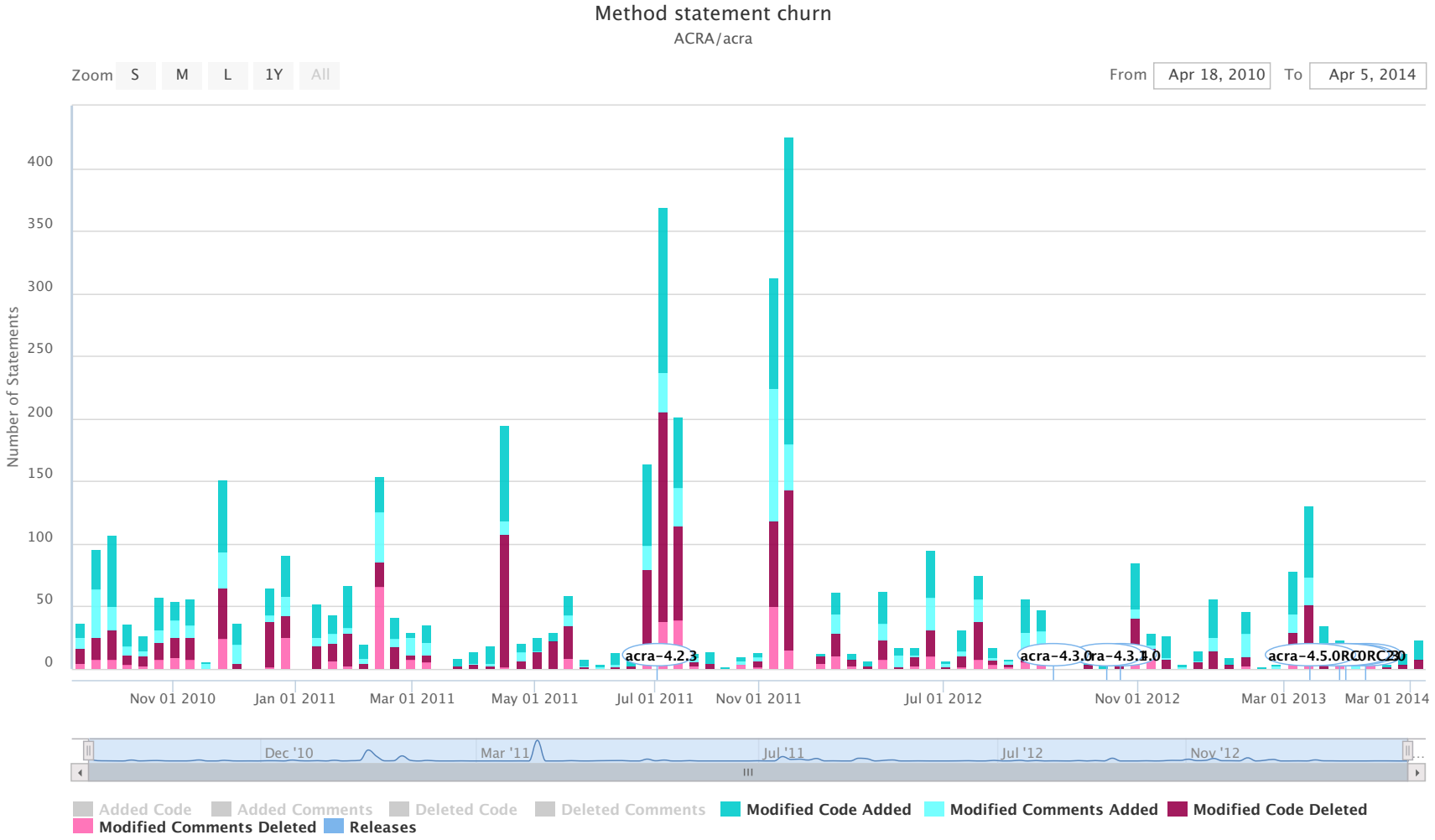
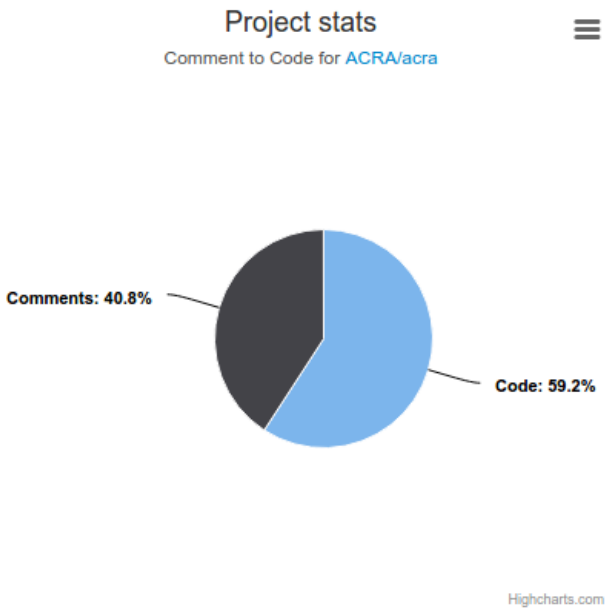


Figure 3.11: Method Statement Modification Visualization for acra

3.4.4 Repository Summary Statistics

For all of the visualizations a summary view was provided which outlines key statistics for the project. In Figure 3.12 the project stats are outlined for *acra*.

- The ratio of comments to code for the history of the project is shown in a pie chart.
- Several table entries outline several top performers metrics are outlined with the top five for each category. The value is shown with the developer's name in square brackets.
    1. top contributers for adding source code
    2. top contributers for deleting source code
    3. top contributers for modifying source code
    4. top contributers for adding comments
    5. top contributers for deleting comments
    6. top contributers for modifying comments
    7. top overall contributers ($\sum code_+ - \sum code_- + \sum doc_+ - \sum doc_-$)
    8. top overall committers
- Customizable legend that allows for data sets to be set to hidden or shown.



Top Committers	Top Contributors
1. KevinGaudin [*573*]	1. KevinGaudin [*8569*]
2. William Ferguson [*133*]	2. William Ferguson [*-720*]
3. pyricau [*48*]	3. william-ferguson-au [*341*]
4. william-ferguson-au [*19*]	4. npombourcq [*-42*]
5. npombourcq [*5*]	5. Richard Meinsen [*22*]

Top Coders	Top Modifiers	Top Deleters
1. KevinGaudin [*8388*]	1. KevinGaudin [*2182*]	1. KevinGaudin [*3126*]
2. pyricau [*2469*]	2. William Ferguson [*332*]	2. pyricau [*2469*]
3. William Ferguson [*1826*]	3. npombourcq [*43*]	3. William Ferguson [*2089*]
4. william-ferguson-au [*358*]	4. william-ferguson-au [*29*]	4. Richard Meinsen [*104*]
5. Richard Meinsen [*126*]	5. meddins86 [*13*]	5. william-ferguson-au [*40*]
Top Commenters	**Top Comment Modifiers**	**Top Comment Deleters**
1. KevinGaudin [*5502*]	1. KevinGaudin [*1191*]	1. pyricau [*2219*]
2. pyricau [*2219*]	2. William Ferguson [*80*]	2. KevinGaudin [*1461*]
3. William Ferguson [*840*]	3. meddins86 [*3*]	3. William Ferguson [*1196*]
4. Richard Meinsen [*96*]	4. william-ferguson-au [*2*]	4. Richard Meinsen [*96*]
5. william-ferguson-au [*34*]	5. fernandohur [*2*]	5. william-ferguson-au [*8*]

Figure 3.12: Project Summary Statistics for acra

3.4.5 Method Change Type

In order to provide a more in depth view of the changes that are applied to a repository was created. Unlike the previous three visualizations that showed count of each type of change changes that occurred per commit, this view presents the type of changes that occurred is per method per commit. Method level changes are the primary aspect feature shown similar to the Method Change View in Figure 3.8.

The method change type defines what type of change that occurred for that method, whether newly added, removed or modified. Each of these are defined in Section 3.4.2. The method change type view is shown for acra in Figure 3.13. The view has a table structure where each line is a new row. The graph is divided into two columns with the vertical dashed line dividing these two columns. The first column, the series of numbers is the method unique identifier. The second column is the list of changes the method received per commit. In the example a snapshot of the view was taken and for spacing reasons full length of the second column was cut out. The list present in the second column is very long because it will contain one character per commit. Therefore in the case of acra which has 404 commits, the full column length would be 404.

As noted one character is assigned per commit to identify what type of change was made on that method. The character mappings are as follows:

- ***empty space*** denotes that the given method not existing in the corresponding commit. This may mean the method has not been created yet or was removed from the repository in the current or a previous commit.

- + A plus sign denotes that the given method was created in the corresponding commit.

- = A equal sign denotes that the given method was unchanged in the corre-

sponding commit.

- **-** A minus sign denotes that the given method was not involved but still exists in the repository for the given commit.

- # A hash sign denotes that the given method was modified in the corresponding commit.

An example of the data presented in Figure 3.13 would be the row with a *method_id* of 9. The changes made to this method starts in the first commit where the method is added. In the second commit the method was then changed. The following commit the method was not within the commit but still existing within the repository. Finally later on the method experienced either no change in the commit or was not present in the commit until the 104th commit which contained the methods deletion.

The view is quite detailed and allows for the comparison of the different changes made to a repository over the course of development. The list presented was sorted based on the first commit the method was added to. Therefore methods that were added at the same time will be closer together and available for more direct comparison.

This view is less sophisticated compared to the previous ones the amount of data available and the potential for use. Currently, the view is presented in a textual form with only one sorting method. More work could be done to include a interactive user interface as well as sorting, group and filter features to help improve accessibility of the data.

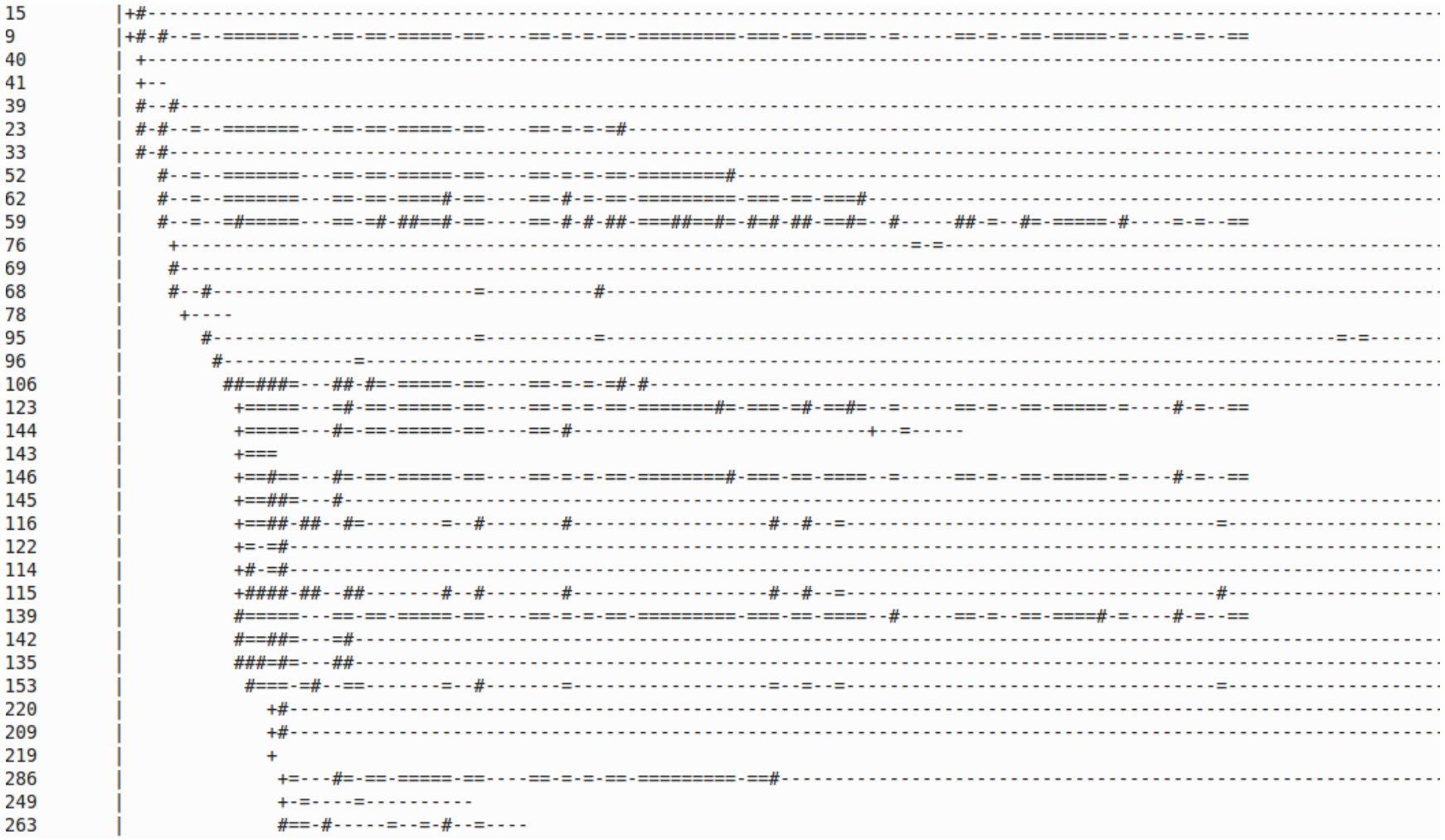


Figure 3.13: Method Change Type for acra

3.5 Visualization Summary

The approach requires repository data to create the prediction model for future changes. The data is extracted from GitHub and stored in a local database. After the collection is complete the data is processed to extract key features and again stored in a local database. This data was explored through several visualizations that all focus on different aspects. The creation and use of these visualizations helped in the definition and selection of the candidate feature (in Table 4.2) used to train the prediction model.

Chapter 4

Prediction with Commit Data

The visualizations, presented in Chapter 3, for the repository data collected provides several insights which can be used to help with the creation of a prediction scheme. With the data visualized, a more general look of the data collected is available. While creating the method for predicting change within the repository, the visualizations provide a helpful resource. The visualization can also help identify relationships between variables and general trends. The actual data used for training the prediction model is outlined in Section 4.1. After that the prediction model is detailed in Section 4.2.

The data presented in the visualization is used towards creating an approach to **predict whether a method will change within the next five commits**. The machine learning algorithms used in the approach are Support Vector Machine (SVM) and Random Forest (RF). We also assess the performance of two prediction methods, SVM and RF with respect to: the size of the sample, the features used for training and balancing of the data set.

4.1 Prediction Data

The data used to predict changes within a repository is the same data used in the visualizations. For more information about the specific information collected see: Sections 3.1 – 3.3. The commit data collected from the target Open Source Software (OSS) repository is used to make predictions. The goal is to predict whether a method within a repository will change in the short term (within the next five commits). By predicting short term changes developers can focus on impending changes rather than changes that may happen in the more distant future. As outlined in Section 3.3, the different types of changes can be either additions, deletions, modifications or no change at all.

4.1.1 Data Training Range

The machine learning model requires data set samples to train from which allows for predictions of new elements. The training samples taken need to be categorized manually and then provided to the machine learning algorithm. This allows for machine learning algorithm to train based on the manual classifications to make future categorizations based on new input data. Since the categorization is whether a method will change, all methods sampled need to be able to collect data the next five commits following the current commit. This requirement provides some restrictions on which values can be included in the sampling. Therefore method changes that are within the last five commits of the training sample window are not included in the training set for the data model, they are instead used to automatically categorize the training data.

As noted above, one of the key factors in the performance of the prediction approach is the size of the sample set. The sample set size is restricted by a variable

Project	**1**	**2**	**5**	**10**
acra	0.43	0.57	0.73	0.73
arquillian-core	0.06	0.1	0.16	0.16
blockly-android	0.4	0.48	0.6	0.6
brave	0.32	0.38	0.47	0.47
cardslib	0.17	0.26	0.39	0.39
dagger	0.51	0.58	0.68	0.68
deeplearning4j	0.29	0.41	0.56	0.56
fresco	0.04	0.07	0.14	0.14
governator	0.34	0.47	0.61	0.61
greenDAO	0.23	0.33	0.45	0.45
http-request	0.73	0.83	0.93	0.93
ion	0.34	0.46	0.67	0.67
jadx	0.15	0.21	0.37	0.37
mapstruct	0.17	0.26	0.38	0.38
nettosphere	0.31	0.44	0.65	0.65
parceler	0.3	0.39	0.53	0.53
retrolambda	0.35	0.46	0.62	0.62
ShowcaseView	0.6	0.74	0.86	0.86
smile	0.13	0.15	0.26	0.26
spark	0.26	0.36	0.53	0.53
storm	0.12	0.22	0.39	0.39
tempto	0.12	0.2	0.35	0.35
yardstick	0.23	0.39	0.62	0.62

Table 4.1: Likelihood of a Method changing in N commits

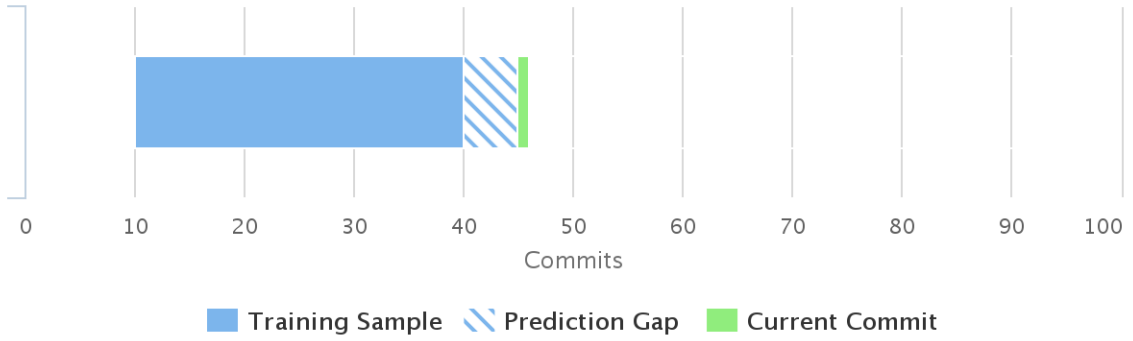


Figure 4.1: Training Sampling Layout

value Sample Window Range (SWR) which controls the number of commits considered to sample from. The data will only be sampled within a limited range of commits as outlined in Figure 4.1. In this case, the SWR is 30 commits and the prediction gap is 5 commits. This 5 commits gap was used for the remainder of our experiments for two reasons:

1. It balanced the sample categories and improves the ability to train the prediction model.

2. It was consistent with our goal of predicting changes that would occur "in the near future.

A larger value l would increase the number of samples likely to change within l commits. Conversely a smaller value, s, would decrease the number of samples likely to change within the next s commits. This is illustrated in Table 4.1. Picking a gap value below 5 would prove difficult to train the model as the number of samples would be drastically unbalanced and require either undersampling or oversampling. Alternatively, picking a gap value above 5 had little impact on the number of samples that contained changes.

4.1.2 Data Distribution

Another consideration when sampling the data is the distribution of the categories. Most of the time, our data set will contain more samples in one category than the other. For example, a sample may contain 80% methods with no change and 20% methods with change in the next five commits. Ideally for training the number of methods with and without changes in the next five commits would be 50%. A bias training data set can adversely affect the model by biasing the predictions towards the over represented category. This is more apparent when the data set category distribution changes drastically between the training set and the testing set. As with our data, this can cause the model to be bias towards one data category and lead to low performance. However in cases where the data is highly skewed to one classification over the other, the model will often predict the larger classification. This can be addressed by using one of the two techniques:

1. Oversampling which will re-samples from the smaller classification to reduce the size difference with the larger classification. Oversampling increases the number of samples by calculating the number of samples in each category and expanding the smaller category by re-sampling values from the data set until both categories are equal in size. Both approaches can be used together so that the smaller category is expanded to at most twice its original size. If the initially smaller category is still smaller, the larger category set will be reduced to the size of the smaller category. When selecting values for re-sampling or removal, a randomized selection process is used to ensure the distribution of the data is preserved.

2. Undersampling which will remove samples from the larger classification to reduce the difference in size with the smaller classification. Undersampling is

applied to a data set by measuring the number of samples in each category. The larger of the two is reduced by discarding samples at random until the data set is the same size as the smaller data set. This will reduce the number of samples used to train the model and may reduce the performance of the model based on the decreased number of samples. In cases where there are a limited number of samples for the smaller category, undersampling may not be ideal.

For example, if category a, with $|a| = 100$ and category b with $|b| = 1000$. Oversampling will be applied to category a since $|a| < |b|$. Therefore a will apply random re-sampled until $|a_n| = |a| \times 2$ or $|a_n| = |b|$. Once one of the conditions is met, oversampling is complete. Next undersampling is applied to the larger category b, where samples are randomly removed from b until $|b_n| = |a_n|$.

Once the categories are balanced then the model is trained on the data. However with large sample sizes, a reduction of the sample set may be necessary. The variable $sample_r$ is the percentage of samples taken from the range. Instead of picking an arbitrary number of samples, we used a ratio to scale based on the number of available samples. When sampling, if the ratio is at 50% then only half of the values retrieved will be used to train or test. For some of the larger data sets sampling 100% of the data from the range would take a lot longer. Therefore sampling a percentage of the data set is commonly used to decrease the training time. In order to provide a more stable model, a random sample of the range is used so that each data entry in the sample has the same chance to be within the training or test data set. With a random selection, each entry in the training sampling space is just as likely to be picked to be included. Therefore at least some useful data entries will be picked. This is ideal compared to a more expensive selection process to attempt to identify useful data entries for selection. Using the example from above, a and b have been oversampled and undersampled such that their new sizes are represented by $|a_n|$ and

$|b_n|$ respectively. Given that a sample ratio of 50% is used then both sets a and b would be reduced by the ratio by randomly sampling from each set to create new sets. The size of each set would be $|a_n| \times r$ where r is the ratio value.

The Table 4.2 outlines each of the considered features used for training the prediction model on. An example of each feature is also provided. As stated in the Section 2.3.3.1, the values need to be processed into a usable format for SVM or RF. First the data is extracted from the database as *raw* values as shown in the ***Data*** column. Text values are mapped to a integer value. For example the *Name* value, "Main.java" will be mapped to the value 3. The reason the value is 3 is because 2 other methods have already been mapped and therefore method name is mapped to the next available mapping. Similarly both *Com* and *Sig* will be mapped from their respective values "void getValue()" and "bob" to 46 and 5. Numerical values are converted by casting the value to a floating point value if the value is not that type already. For spacing reasons, all the values in the table that have no decimal value are shown without a ".0" following.

Several experiments were conducted to investigate the benefit of each of the candidate features. The 17 candidate features were narrowed down from this initial list into a smaller set of 7 training features. These feature sets were constructed to determine potential ideal feature sets. The complete list of feature sets used are shown in Table 4.3. This list is not listed by execution order but rather the first five are kept consistent with the numbering used in Chapter 5. Some of the feature sets are not full explained in the table for spacing reasons. Feature sets 16, 17, 18 and 27 all use the last five previous changes or durations. Each are marked similarly to those that only use the most recent previous change or duration except for a footnote marker. Likewise feature set 29 is also different since the previous change used is only the change made five commits ago.

Feature	**Description**	**Data**	**Example Vector**
Com	The individual who committed the change	bob	5
Sig	The method signature related to the change details	void getValue()	46
Name	The name of the file	Main.java	3
Δ_i	Whether the method changed or not in the current commit	3	1
m_+	Whether the is newly added	3	0
m_-	Whether the method was deleted	3	0
m_c	Whether the is a modification	3	1
m_x	Whether the received no change	3	0
Δ_{i-j}	Whether the method changed in a previous commit	0	0
$type_{\Delta_{i-j}}$	Type of method changed in a previous commit	2	2
f_Δ	The frequency that the method is changed within the SWR	0.0464	0.0464
sf_Δ	The frequency that the method is changed within the last 10 commits.	0.1	0.1
t_Δ	The time between the current commit c_i and the previous commit c_{i-1}	2148	2148
$t_{\Delta_{i-j}}$	The time difference between a sequence of two previous commits	453	453
Length	The length of the method in this commit	10	10
$change_{t-1}$	Whether a change has occurred in the previous 5 commits	$\{3, 0, 0, 3, 0\}$	1
$change_t$	Identifies whether a change occurred within the next 5 commits for the given method	0	0

Table 4.2: Candidate features for SVM model

Each of these feature sets are tested on the repository acra and the results are shown in Figure 4.2. Clearly feature sets 1, 2, 4 and 5 all perform better overall with high performance for the three measures:

- Precision
- Recall
- Accuracy

Feature set 3 did not perform as well as the other selected feature sets but did have a high recall value. The goal was to select a feature set that did not perform as well for acra but still had fairly high performance to potentially perform better for other projects. For a number of cases the feature set performed well for one or two of the three performance measure but performed poorly in the rest. Therefore the first five

Feature Sets	**Com**	**Sig**	**Name**	Δ_i	m_+	m_-	m_c	m_x	Δ_{i-j}	$type_{\Delta_{i-j}}$	f_Δ	sf_Δ	t_Δ	$t_{\Delta_{i-j}}$	**Length**	$change_{t-1}$
1	•	•	•								•	•			•	•
2	•	•	•								•		•		•	•
3	•	•	•								•		•			•
4		•	•								•		•			•
5	•	•	•								•					•
6	•	•	•	•							•					•
7	•	•	•		•						•					•
8	•	•	•			•					•					•
9	•	•	•				•				•					•
10	•	•	•					•			•					•
11	•	•	•						•		•		•			•
12	•	•	•							•	•		•			•
13	•	•	•							•	•					•
14	•	•	•						•		•					•
15*	•	•	•								•			•		•
16†	•	•	•						•		•					•
17‡	•	•	•							•	•					•
18*‡	•	•	•							•	•		•			•
19	•	•	•		•	•	•				•					•
20	•	•	•			•	•				•					•
21	•	•	•			•		•			•					•
22		•	•								•					•
23		•	•								•					
24		•	•						•		•					
25		•	•	•							•					
26		•	•	•							•		•			
27†		•	•	•					•		•					
28		•	•	•					•		•					
29§		•	•	•					•		•					

Table 4.3: Training Features

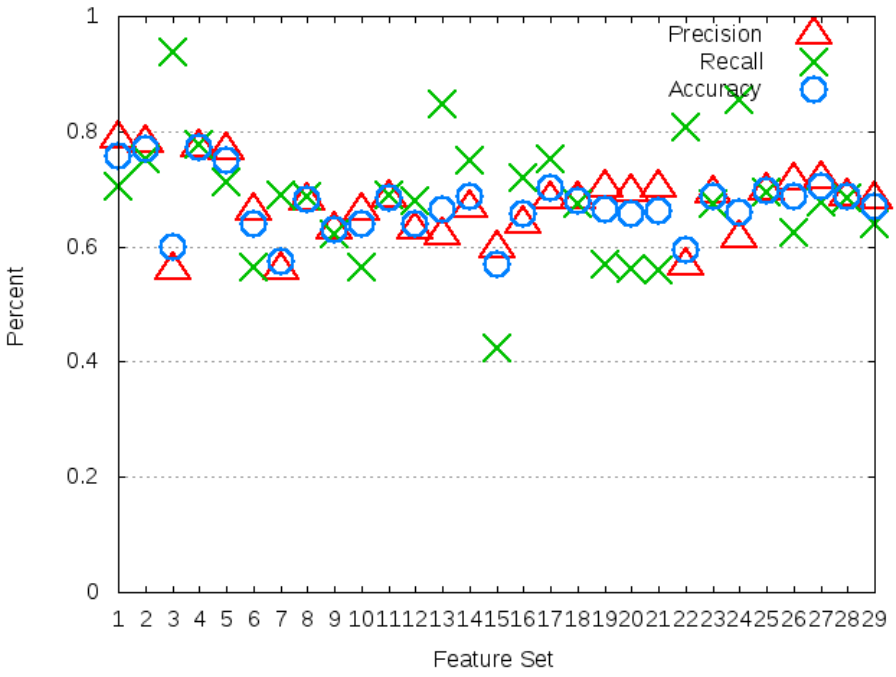


Figure 4.2: Feature Sets Analysis using RF

were selected for experimentation of the effect of feature sets on various projects using SVM and RF.

Another small change made to the data to create a vector for the prediction model was to convert the change type into a change indicator vector using Equation 4.1. The vector is converted into a single value which indicates whether a change has occurred in the previous five commits. This process is done through calculating the sum of the change vector using Equation 4.2. Finally, the change indicator, $change_{i-1}$, is identified using Equation 4.3.

$$c_i = \begin{cases} 1 & \text{if } \Delta_i > 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

*$t_{\Delta_{i-j}}$ denotes the use of the 5 previous change duration for this feature set.
†Δ_{i-j} denotes the use of the 5 previous changes for this feature set.
‡$type_{\Delta_{i-j}}$ denotes the use of the 5 previous change types for this feature set.
§Δ_{i-j} denotes whether change occurs at the fifth previous commit for this feature set.

$$reduce = \sum_{i=t-5}^{t} c_i \tag{4.2}$$

$$P = \begin{cases} 1 & \text{if } reduce > 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

f_Δ is calculated by taking the number commits in which involve changes to the current method (c_i) within the SWR divided by the current number of commits (c_{cur}) since the start of the SWR. This is formalized in Equation 4.4. A frequency of change is available for the duration of the SWR.

$$f_\Delta = \frac{|c_i|}{|c_{cur}|} \tag{4.4}$$

sf_Δ is calculated by reducing the range sampled to s. Then counting the number of times the method changes within the last s commits and dividing it by s. The size of the short frequency can be any value that is less than the size of the SWR. For use in the rest of the paper $s = 10$ which means that the sf_Δ is for the last 10 commits.

t_Δ is the difference between the current commit time ($t(c_i)$) and the previous commit time ($t(c_{i-1})$) calculated in Equation 4.5. Both time values are provided as time stamps and the result is calculated in seconds. Only the difference in time between the current commit and the previous one is calculated therefore, in Equation 4.5, i denotes the current commit.

$$\Delta t_i = t(c_i) - t(c_{i-1}), i > 1 \tag{4.5}$$

4.2 Prediction Method

For our prediction problem, a machine learning algorithm is used to create a prediction model. The data used to train the model is collected as shown in Section 4.1. The machine learning algorithms that can be used are either SVM or RF (see Section 2.3.3.1 and Section 2.3.3.2 for more details). Each of these method are widely used for data mining techniques and are easy to use. Figure 1.1 outlines the overall structure of the approach for how changes are predicted.

The SVM model was created through the use of a libsvm[1] binding for Ruby, rb-libsvm[2]. This library was a good fit since the data was collected using a Ruby script. The rb-libsvm library facilitated the creation, training and testing of the model. For the RF model, a Python library scikit-learn[3] was used. Python was used instead of Ruby because of a lack of a mature RF library in Ruby. The scikit-learn library provided creation, training, testing and feature importance weight. Both libraries have explicit and implicit restrictions on the data. The explicit restrictions related to data format which is deemed unacceptable. Implicit restrictions are expected to be in forced and failure to do so leads to poor prediction results. Further information about the preprocessing of the data can be found in Section 4.1.2. After processing the data, the respective libraries were given the data. In the case of RF the feature importance weights are calculated throughout the model training and allow for introspection on the training features.

Use of the approach requires a few steps in total before predictions can begin. Firstly, the data related to the project must be collected from GitHub. Once the data is collected, the prediction model can be created by providing training set built

[1] `https://www.csie.ntu.edu.tw/~cjlin/libsvm/`
[2] `https://github.com/febeling/rb-libsvm`
[3] `http://scikit-learn.org/stable/`

from sampling the data set. After training the model, predictions can be made using the model on new data. The next chapter analysis the approach by training and testing the model using subsets of the project data sets.

Chapter 5

Experiments

Experiments were conducted in order to validate for the proposed approaches. The goal of these experiments is to determine whether data from an Open Source Software (OSS) repository can predict changes that occur within the next 5 commits. These experiments are based on the approach outlined in the previous chapter, Chapter 4. The experiment was conducted through measuring the performance of each core factors varied in isolation. Specifically the Sample Window Range (SWR), the feature set and categorization balancing using oversampling. These three factors are explored for both machine learning algorithms.

5.1 Experimental Repository Data

The complete list of repositories used in the experiment data are found in Table 5.1. Commit data from each repository was collected from the creation date for the repository till the data collection date. The commit data excludes any commit that lacked a change to a file containing Java code. Since the primary interest was to parse Java code, files containing Java code were used while all other files are ignored. These

measures provide a more accurate description of the repository in terms of the analysis and predictions made on it. Secondly, the number of developers does not map effectively to what git uses as committers and authors. Instead, the number of developers includes all individuals (removing duplicates) who committed or authored commits to the current repository.

Owner	**Repository**	**Start Date**	**End Date**	**# of Commits**	**# of Developers**
ACRA	acra	2010-04-18	2015-06-05	404	32
arquillian	arquillian-core	2009-11-13	2016-03-16	473	49
google	blockly-android	2015-07-23	2016-06-23	691	8
openzipkin	brave	2013-04-07	2016-06-21	337	32
gabrielemariotti	cardslib	2013-09-20	2015-05-12	327	13
square	dagger	2012-06-25	2016-01-30	496	38
deeplearning4j	deeplearning4j	2013-11-27	2016-02-13	3523	61
facebook	fresco	2015-03-26	2015-10-30	313	45
Netflix	governator	2012-03-18	2016-06-23	621	31
greenrobot	greenDAO	2011-07-28	2016-05-23	415	4
kevinsawicki	http-request	2011-10-21	2015-01-21	273	14
koush	ion	2013-05-22	2016-06-14	520	29
skylot	jadx	2013-03-18	2016-03-27	480	11
mapstruct	mapstruct	2012-05-28	2016-06-15	604	22
Atmosphere	nettosphere	2012-02-09	2016-04-11	336	12
johncarl81	parceler	2013-07-03	2016-06-22	228	12
orfjackal	retrolambda	2013-07-20	2016-04-30	275	11
amlcurran	ShowcaseView	2012-08-14	2016-05-30	332	39
haifengl	smile	2014-11-20	2016-06-24	237	14
perwendel	spark	2011-05-05	2016-06-19	551	86
apache	storm	2011-09-16	2015-12-28	2445	260
prestodb	tempto	2015-03-06	2016-06-20	298	19
gridgain	yardstick	2014-04-11	2015-10-12	213	12

Table 5.1: OSS Repositories used in Experiments

1. **acra**[1] is an Android bug logging tool used with Android applications to capture information related to bugs or crashes. The information is sent to the developers

[1] `https://github.com/ACRA/acra`

to help them address the issues that their clients encounter while using there application.

2. **arquillian-core**[2] is a platform for creating automated integration, functional and acceptance tests for Java middleware products.

3. **blockly-android**[3] provides a native implementation of the blockly library for drag and drop development on Android.

4. **brave**[4] provides a Java distributed tracing tool for troubleshooting latency problems and is compatible with Zipkin.

5. **cardslib**[5] is an Android library for creating UI Cards in an Android application.

6. **dagger**[6] from square is a Java application used to satisfy dependencies for classes to replace the factory model of development.

7. **deeplearning4j**[7] is a distributed neural network library that integrates Hadoop and Spark. This application is the largest of the all the repositories and provides a large wealth of data to analyze.

8. **fresco**[8] from facebook is the smallest repository with the shortest development period. This repository provides a library for using images on Android to attempt to solve limited memory issues with mobile devices.

9. **governator**[9] is a library of extensions and utilities that enhances Google's Guice to provide injector life-cycle and object life-cycle.

[2] `https://github.com/arquillian/arquillian-core`
[3] `https://github.com/google/blockly-android`
[4] `https://github.com/openzipkin/brave`
[5] `https://github.com/gabrielemariotti/cardslib`
[6] `https://github.com/square/dagger`
[7] `https://github.com/deeplearning4j/deeplearning4j`
[8] `https://github.com/facebook/fresco`
[9] `https://github.com/Netflix/governator`

10. **greenDAO**[10] provides an Android based light and fast object relational mapping to SQLite database entries.

11. **http-request**[11] is a library accessing the *httpURLConnection* to make requests and then access the response.

12. **ion**[12] provides asynchronous networking and image loading for Android.

13. **jadx**[13] is a Java decompiler for Android Dex and Apk files.

14. **mapstruct**[14] is an annotation processor for generating type-safe bean mapping classes.

15. **nettosphere**[15] provides a WebSocket/HTTP server based on Atmosphere and Netty Framework.

16. **parceler**[16] is a library for creating serialize code.

17. **retrolambda**[17] provides a backport for lambda expressions implemented in Java 8 to Java 7, 6 and 5.

18. **ShowcaseView**[18] is a library for Android that can highlight and showcase components within the UI of a application.

19. **smile**[19] stands for Statistical Machine Intelligence and Learning Engine and is a machine learning library for Java.

[10] `https://github.com/greenrobot/greenDAO`
[11] `https://github.com/kevinsawicki/http-request`
[12] `https://github.com/koush/ion`
[13] `https://github.com/skylot/jadx`
[14] `https://github.com/mapstruct/mapstruct`
[15] `https://github.com/Atmosphere/nettosphere`
[16] `https://github.com/johncarl81/parceler`
[17] `https://github.com/orfjackal/retrolambda`
[18] `https://github.com/amlcurran/ShowcaseView`
[19] `https://github.com/haifengl/smile`

20. **spark**[20] a tiny web framework for Java 8.

21. **storm**[21] from apache real time computational system for continuous streams of data. This repository is one of the larger repositories and has a large development community.

22. **tempto**[22] A testing framework for SQL databases running on Hadoop.

23. **yardstick**[23] is a framework for creating benchmarks specifically for clustered or distributed systems.

Given the large number of repositories, a categorization system was established to group repositories based on similar attributes. Four repository measures were selected for comparing the repositories and are outlined in Table 5.2. The measures are:

- Repository length in years.
- Repository size in number of methods.
- Number of developers.
- The rate of commits made in commits per year.

The repository length represents the number of years the repository has been under development for. The size of the repository is measured in the number of method signatures within the repository since created. The number of developers is tallied from the beginning of the repository for this measure. Finally, the rate of commits is the number of commits contributed to the repository per year. A yearly rate of commits was sufficient since the majority of the repositories had more than one year of development history.

[20] `https://github.com/perwendel/spark`
[21] `https://github.com/apache/storm`
[22] `https://github.com/prestodb/tempto`
[23] `https://github.com/gridgain/yardstick`

Repo Duration	**Repo Size**	**# Devs**	**Commit Rate**
short ($t < 1$)	small ($m < 2000$)	small ($d < 30$)	low ($r < 100$)
medium ($1 \leq t < 3$)	medium ($2000 \leq m < 10000$)	medium ($30 \leq d < 100$)	medium ($100 \leq r < 300$)
long ($t \geq 3$)	large ($m \geq 10000$)	large ($d \geq 100$)	high ($300 \leq r < 600$)
			very high ($r \geq 600$)

Table 5.2: Experiment Repository Summary

Using the classifications outline in Table 5.2, the repositories are grouped and organized with similar repositories. In Table 5.3 the repositories are sorted by their classification and have dividing lines around similar repositories. For example four repositories; http-request, nettosphere, parceler and retrolambda are all classified in the same group. Some repositories like ion or storm are not grouped in with another repository and thus are in a group of their own.

Each of the repositories are selected from GitHub using the list of Java repositories with a large amount of development. OSS repositories were targeted to simplify any usage concerns. Specifically, OSS repositories are open and freely available immediately and can be discussed without restriction. Therefore in order to be selected the program had to clearly use an OSS license. Secondly, the repository also needed to have at least a 6 months worth of development and at least 300 commits to provide a large enough dataset to analyze. An effort was also made to pick repositories of different sizes to provide better tests of various conditions.

In order to get a more detailed understand of the selected repositories, numerous measures were taken. These measures also allow for each repositories to be compared to each other in terms of the development of each of the repositories. For example the size of the repository is represented through several measures including: number of commits, methods and developers. Several averages are calculated to help establish how the development occurred within a repository during the development. A few

Repo Name	Repo Duration	Repo Size	# Devs	Commit Rate
yardstick	medium	small	small	medium
tempto	medium	medium	small	medium
blockly-android	medium	medium	small	high
fresco	medium	medium	medium	high
http-request	long	small	small	low
nettosphere	long	small	small	low
parceler	long	small	small	low
retrolambda	long	small	small	low
ion	long	small	small	medium
acra	long	small	medium	low
dagger	long	small	medium	low
ShowcaseView	long	small	medium	low
greenDAO	long	medium	small	low
smile	long	medium	small	low
cardslib	long	medium	small	medium
jadx	long	medium	small	medium
mapstruct	long	medium	small	medium
arquillian-core	long	medium	medium	low
brave	long	medium	medium	low
spark	long	medium	medium	low
governator	long	medium	medium	medium
deeplearning4j	long	large	medium	very high
storm	long	large	large	high

Table 5.3: Experiment Repository Summary

Repository	**# of Methods**	**# of Methods Changes**	**Avg # of Commits / Year**	**Avg # of Methods Change / Commit**
acra	1309	3605	67.33	9.51
arquillian-core	5563	6657	59.13	15.2
blockly-android	3608	9679	345.5	14.82
brave	4204	7823	84.25	26.98
cardslib	3940	5122	109.0	16.68
dagger	1827	6314	99.2	13.7
deeplearning4j	29896	82198	880.75	24.33
fresco	3463	4139	313.0	14.73
governator	4229	10946	124.2	19.04
greenDAO	4089	8625	69.17	21.84
http-request	726	1740	54.6	6.72
ion	1678	4347	130.0	8.82
jadx	6012	9322	120.0	19.63
mapstruct	7885	10185	120.8	19.04
nettosphere	1112	2857	67.2	9.01
parceler	1619	3076	57.0	14.72
retrolambda	1111	2588	68.75	9.95
ShowcaseView	927	2672	66.4	8.62
smile	3885	3879	79.0	18.47
spark	3117	9154	91.83	18.27
storm	14599	50037	489.0	24.03
tempto	2422	3386	149.0	11.96
yardstick	512	1216	106.5	6.37

Table 5.4: Repository Change Statistics I

Repository	Avg # of Methods Change / Year	Avg # of Changes / Method	Avg # of Commits / Developer	Max Commits / Year	Min Commits / Year
acra	600.83	4.52	13.93	119	33
arquillian-core	832.13	2.03	36.38	175	6
blockly-android	4839.5	4.68	98.71	690	1
brave	1955.75	4.24	14.65	108	56
cardslib	1707.33	3.28	46.71	223	3
dagger	1578.5	5.64	16.0	236	4
deeplearning4j	20549.5	5.69	65.24	2018	65
fresco	4139.0	1.49	156.5	313	313
governator	2189.2	4.11	24.84	159	75
greenDAO	1437.5	3.94	138.33	137	5
http-request	348.0	2.56	39.0	108	5
ion	1086.75	3.31	40.0	253	7
jadx	2330.5	2.41	43.64	208	11
mapstruct	2037.0	2.04	54.91	288	7
nettosphere	571.4	4.37	37.33	118	5
parceler	769.0	2.43	45.6	76	41
retrolambda	647.0	3.06	25.0	133	24
ShowcaseView	534.4	5.9	10.71	141	6
smile	1293.0	1.86	19.75	121	6
spark	1525.67	3.92	7.25	171	22
storm	10007.4	5.93	15.47	948	118
tempto	1693.0	1.88	16.56	253	45
yardstick	608.0	3.65	23.67	208	5

Table 5.5: Repository Change Statistics II

examples of average measurements are the number of commits per year, changes per method.

Several average measures were also taken which detail the amount of change that occurs within the repository. The average number of commits per repository coupled with the average number of changes per commit clearly indicates the amount of changes that are occurring with in the repository. The rate at which methods are change provides good insight into the growth of a repository. While some changes

may involve the addition of new methods, others may include the removal of methods or the modification of methods. The other measures relating to the amount of change occurring with a repository on average are the number of methods changed per year and the number of changes per method. Each of these further outline how the changes are being made to the repository on average.

A few of the measures are related to the number of developers. These while provided are not the primary focus. The information provided by tracking developer interactions with each other or the repository could be integrated into future work.

While the purposed method was being developed ACRA's acra repository was primarily used for exploring and initial testing of the approach. After experimenting on acra a few of the potential candidate feature sets were distinguished based on their superior performance. Experiments were then run on other repositories using the feature sets that performed better.

5.2 Experimental Setup

The experimental setup defines numerous parameters of different importance to conduct experiments. The majority of these parameters will remain constant to help observe the impact differences in the independent variable will have on the three dependent variables; precision, recall and accuracy. Each experiments will use one of the parameters as the independent variable. The three independent variables used in the experiments are outlined below. An experiment consists of a set of trails where the independent variable is modified to measure the resulting dependent variables. The results of a single trial or of the set of trails for a repository will be referred to as the performance of the approach. In order to reduce specific repository confounding factors numerous repositories were tested on. This will be discussed further

Repository	**Max # of Methods Changed / Year**	**Min # of Methods Changed / Year**	**Max # of Change / Method**	**Max # of Commits / Developer**	**Min # of Commits / Developer**
acra	1503	183	52	229	1
arquillian-core	3421	55	110	420	1
blockly-android	9543	136	90	538	1
brave	3300	1038	83	225	1
cardslib	3340	19	95	285	1
dagger	3374	171	65	157	1
deeplearning4j	35869	4377	345	1987	1
fresco	4139	4139	33	269	44
governator	3324	1066	263	316	1
greenDAO	2971	34	63	367	1
http-request	752	14	50	267	1
ion	2315	24	161	492	1
jadx	3915	248	197	436	1
mapstruct	4462	201	81	334	1
nettosphere	1074	10	46	322	1
parceler	1151	516	31	217	1
retrolambda	1501	212	83	237	1
ShowcaseView	1156	74	70	215	1
smile	1918	872	24	155	1
spark	2818	277	72	277	1
storm	26526	2152	314	622	1
tempto	3073	313	44	66	1
yardstick	1163	53	62	137	1

Table 5.6: Repository Change Statistics III

in the discussions section. The experiments section only includes a small sample of the repositories that were experimented on. The repositories shown are ones that exhibited interesting patterns or results. The complete set of performance figures for every repository are shown in Appendix A.

This thesis works to determine whether Support Vector Machine (SVM) or Random Forest (RF) can be used to effectively predict changes that will occur within the repository. To potentially provide an answer to this question the factors that are used for the prediction method are studied. The experiments attempt to determine what impact the different factors will have on the purposed methods. These factors include:

1. The SWR which is the size of range which the samples are taken from.

2. The set of features used to train the machine learning model.

3. The distribution of the data through use of oversampling.

Through investigating these factors a more clear picture of the performance of the approach will be be provided. Without such a investigation the method contains could produce capable solution just as likely as poor solutions. Worse still, the setup may produce poor solutions more often than capable solutions. Once a more concrete understanding is developed of the different factors and the performance of the algorithm accordingly the research question can be answered as to whether it is possible to predict changes within a repository using the commit data.

5.2.1 Prediction Features

The experimental design to allow for the predictions made on historical data to be tested with available data. Therefore within the data collected for the repository the

predictions must be made for values that are already known to allow for verification. Therefore the experimental sampling would build off of the prediction sampling outlined in Section 4.1. A second region defined as the prediction sample range. The Figure 5.1 outlines the updated layout. The size of the training range ($|s_t|$) and the size of the prediction ($|s_p|$) are able to be different sizes, however for each experiment they remain the same ($|s_t| = |s_p|$).

The sample range is taken from the current commit c_i to c_{i-g-m} in the case that $i > m$. m denotes the size of SWR in commits and g is the number of commits the change is predicted within. For example if the model is predict a change that occurs within the next 5 commits ($g = 5$) and $m = 30$ then Figure 5.1 shows how the data would be sampled. The training sample would be where data would be collected from to train the model. The prediction gap is to account for the data sampling calculating whether methods at commit 40 will have a change within the next 5 commits. Therefore to properly test it on data that is not used as part of the testing model the offset is needed. The SWR for the testing data set is labeled as the *Testing Sampling*.

The sliding window factor is one of core aspects related to extracting samples from the data set. When using the sliding window to sample the data the data is divided as shown in Figure 5.1. The training sample is where the training data set is sampled from. The testing sample is where the testing data is sampled from.

A data set with an extended sampling range will extend the sampling range beyond the original size for either the training sample or the testing sample. The training range can be expanded to include earlier samples to increase the sample space.

The training and testing sampling range are defined as the number of commits from which the samples can be taken. In Figure 5.1, both the training and testing sample ranges are set to 30 commits. These two values can differ from one another

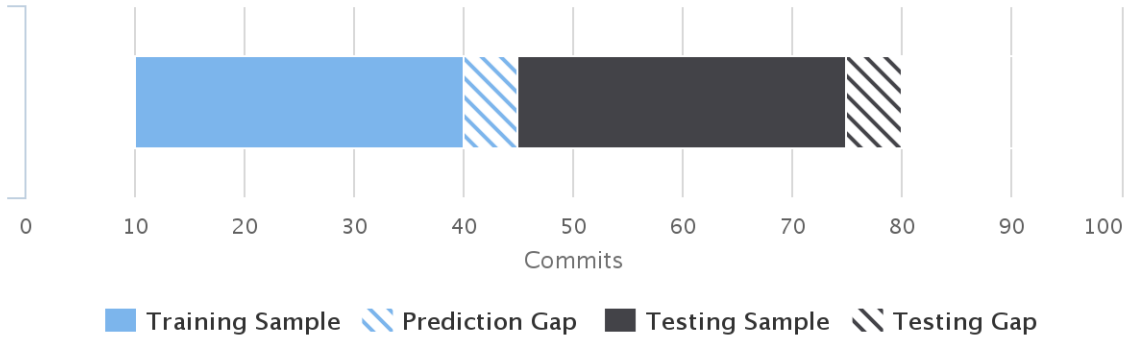


Figure 5.1: Sampling Window Layout

but tend to be kept the same for most of the experiments.

As discussed in Section 4.1, sample biasing can cause the distribution to favor the selection of one category over another. Undersampling and oversampling can prevent the model from simply classifying all samples as one category or the other.

For each repository data set there are numerous windows that be can be used. The window number is setting which window is used broadly mapping to the position within the data set that the model will be trained on and then predicted on. In Figure 5.1 the *current commit* is located at 45. This is the point from which predictions will be made after. The gap preceding the starting point is is 5 commits long and is followed by the sample window for the training data which is 30 commits long. To calculate the window offset simply using the starting position (p_s), the gap length (g), and the SWR can be calculated in Equation 5.1. Therefore in this case the window offset would be $45 - 5 - 30$ which is 10.

$$wo = p_s - g - swr \tag{5.1}$$

Finally, the last factor of note is the parameters used to configure each prediction method. RF use a single parameter, the size of the forest. SVM meanwhile uses two parameters; C and gamma. Picking the most suitable parameters is ideal to achieve

good performance from the prediction model. For SVM a grid search technique is provided by the developers of the libsvm source[24] for optimizing the parameters. For RF, the size of the forest will have an impact but is far more manageable since it is a single parameter. A larger number of trees in the forest will generally provide better results, but will cause the algorithm to take longer to train.

5.2.2 Prediction Performance

For each experiment where the used random sampling the experiment was performed 5 times to account for variations in the random sample. Therefore if the initial results using the first sample set were not characteristic of the full dataset then running the experiment with more random samples is more likely to represent the true characteristics of the dataset. This required taking five random samples from each quarter, training the model and running the tests on the model to then determine the average prediction score.

The goal of the prediction methods are to provide a good prediction of whether the a given vector will fit in one category or the other. A model's prediction performance can be rated using three measures of accuracy, precision and recall. Accuracy is measured as how often predictions p_i are classified correctly where a_i represents vector v_i correct classification. The prediction accuracy ($P_{accuracy}$) can then be calculated using Equation 5.6. This simply sums up the accuracy for each vector and then divides it by the total number of vectors (where $n = |v|$).

$$tp = \sum_{i=0}^{n} \begin{cases} 1 & \text{if } p_i = a_i \ \& \ a_i = 1 \\ 0 & \text{otherwise} \end{cases} \tag{5.2}$$

[24] `https://www.csie.ntu.edu.tw/~cjlin/libsvm/`

$$tn = \sum_{i=0}^{n} \begin{cases} 1 & \text{if } p_i = a_i \ \& \ a_i = 0 \\ 0 & \text{otherwise} \end{cases} \tag{5.3}$$

$$fp = \sum_{i=0}^{n} \begin{cases} 1 & \text{if } p_i \neq a_i \ \& \ a_i = 0 \\ 0 & \text{otherwise} \end{cases} \tag{5.4}$$

$$fn = \sum_{i=0}^{n} \begin{cases} 1 & \text{if } p_i \neq a_i \ \& \ a_i = 1 \\ 0 & \text{otherwise} \end{cases} \tag{5.5}$$

$$P_{accuracy} = \frac{tp + tn}{tp + tn + fp + fn} \times 100 \tag{5.6}$$

The precision of a model is the measure of how correct the model predicts that a change will occur when it predicts that a change will occur. Given the true positives tp, represents the number of predictions that the model correctly identified as having a change and the false positives fp is the number of times the model incorrect predicted a change to occur when it in fact did not. The equation for calculating precision is show in Equation 5.7.

$$P_{precision} = \frac{tp}{tp + fp} \tag{5.7}$$

The recall of the model is the measure of how correct the model predicts that change will occur out of all the times changes really occurred. Again using tp as the number of true positives, and false negatives fn which is the number of times the model fails to predict that a change will occur. The recall can be calculated using the Equation 5.8.

$$P_{recall} = \frac{tp}{tp + fn} \tag{5.8}$$

5.3 Experimental Results

For each experiment all of the data used to train and test the model is collected using a Ruby script to query a PostgreSQL database. The PostgreSQL database provides the raw data which is then processed into data vectors in an acceptable form for SVM or RF. The data processing method is outlined more completely in Section 4.2.

5.3.1 SVM Experiments

For this set of experiments the machine learning algorithm SVM is used to provide the change predictions. As noted in Section 4.2, the implementation for SVM is a Ruby binding of the original library. The parameters used for all of the experiments with SVM are $C = 10$ and $gamma = 8$.

5.3.1.1 Window Range Experiments

In this experiment the independent variable is the size of the SWR in commits. For each variation of the SWR the performance is measured. In Table 5.7, the features used by the prediction model are outlined. Features with a mark, •, are used while those without are not. In this experiment only the sf_Δ is not used while all the rest are. Each of these features is outlined in further detail in Table 4.2.

Com	Sig	Name	f_Δ	sf_Δ	t_Δ	Length	$change_{t-1}$
•	•	•	•		•	•	•

Table 5.7: SWR Experiment Features

As noted above the independent variable for this experiment is the SWR. The remaining parameters for the experiment are constant for each test. These parameters for this experiment are outlined in Table 5.8.

Extended Window	Over Sampling	Under Sampling	Sample Rate	Window Offset	SVM C	gamma
No	No	Yes	100%	5	10	8

Table 5.8: SWR Experiment Setup

Each repository was tested on using these outlined parameters for an SWR varying from 60 to 130 by intervals of 10. The results for the experiments are shown with the precision, recall and accuracy. For each graph the independent variable is the number of commits in the SWR. Y-axis is the percentage for either the precision, recall or accuracy. The complete set of experimental performance results are found in subsection A.1.1. For some repositories did not have enough data to complete the entirety of this experiment. For example, smile did not have enough data to complete the trials with SWR for 120 or 130. These repositories were still included and show how the method works with smaller amount of data available.

The majority of the repositories using SVM did not perform well with accuracy and precision typically between 0.4 and 0.6. This repositories as well as others have very poor results and show the difficulty of this problem. Similarly, Figure 5.3 shows low precision and accuracy while very high recall. The independent variable, SWR has very little impact on the performance for this repositories in particular.

Some of the repositories like http-request in Figure 5.4 had a large amount of variation with the changes to the SWR. In one case http-request moderately well in SWR 80 while at 60 and 120 the accuracy and recall are 0.

In Figure 5.5, the repositories acra is shown with the best result for SVM. When the SWR is at 70-100 the performance is high, for the other cases the performance is lower but not by a large margin. The point of interest is that recall performs well for an SWR of 100 or lower but performs worse than the accuracy and precision after 100.

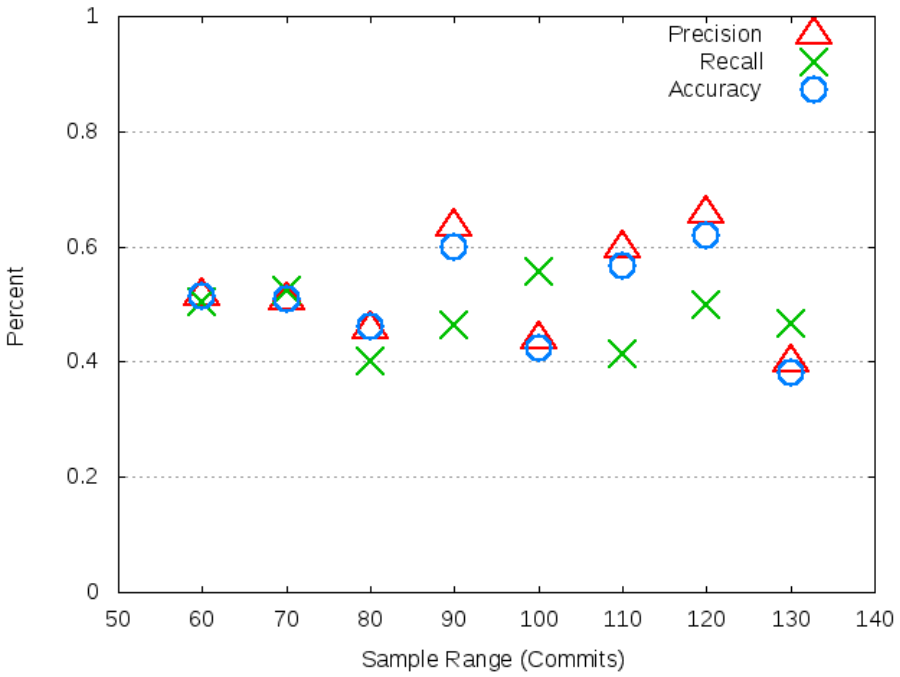


Figure 5.2: SWR for tempto using SVM

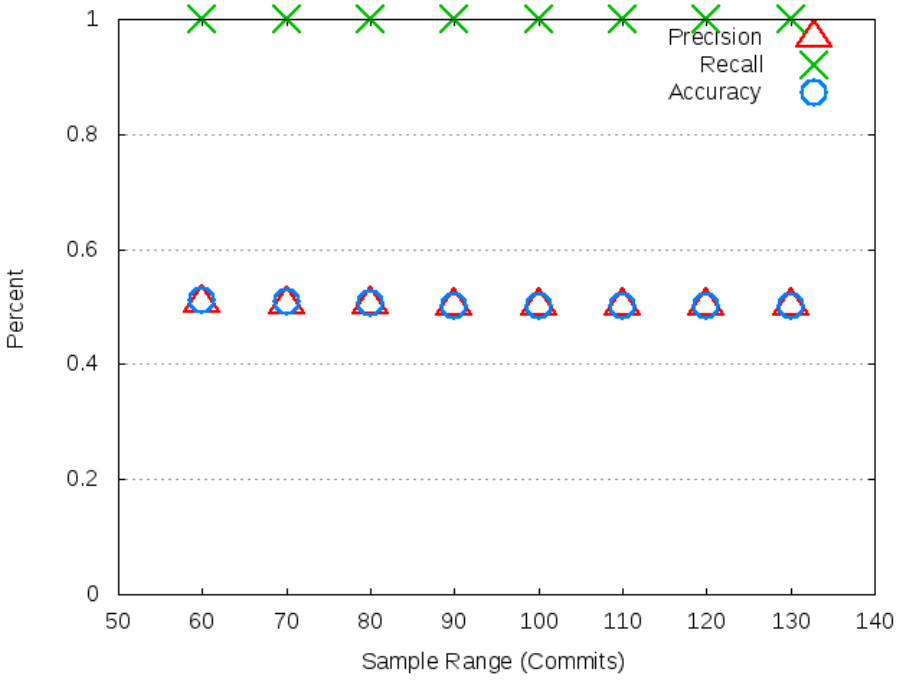


Figure 5.3: SWR for blockly-android using SVM

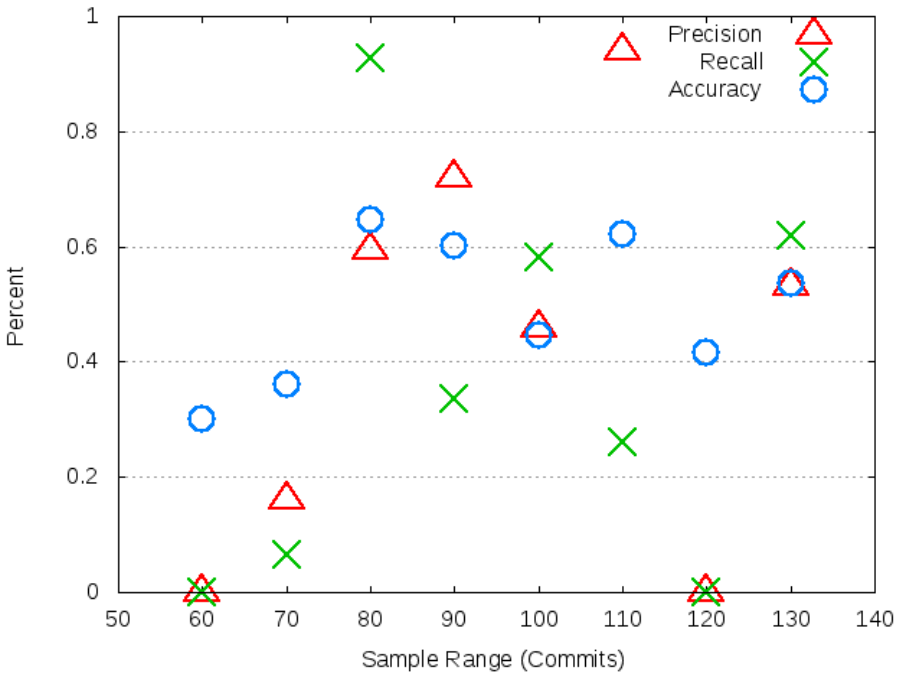


Figure 5.4: SWR for http-request using SVM

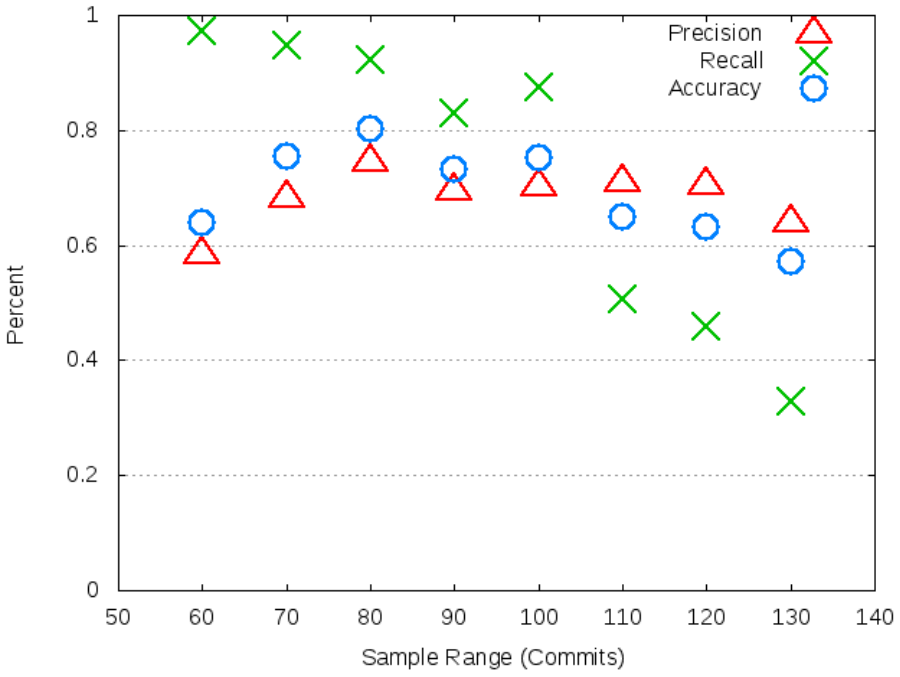


Figure 5.5: SWR for acra using SVM

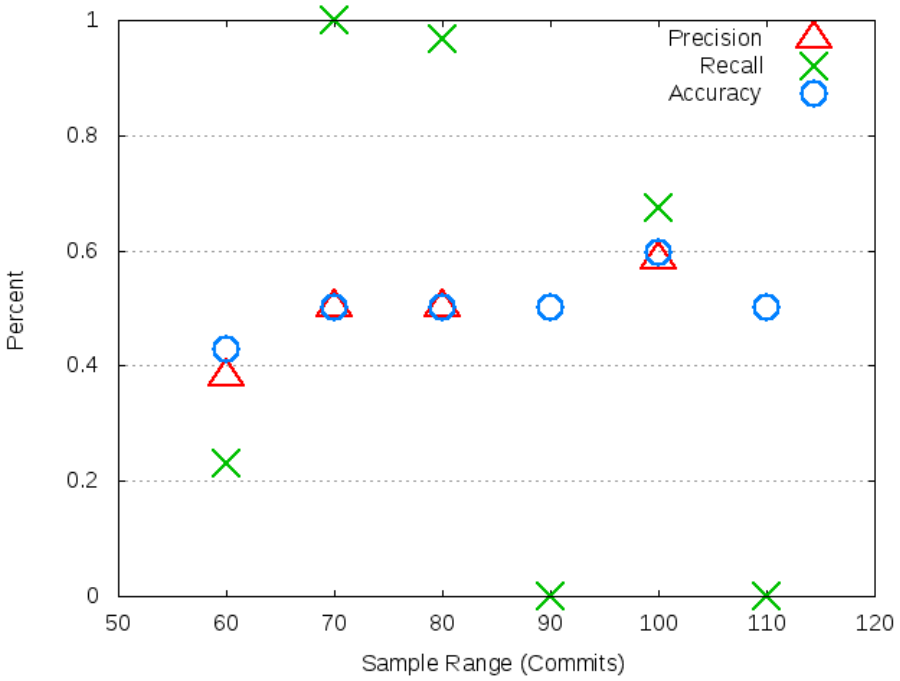


Figure 5.6: SWR for smile using SVM

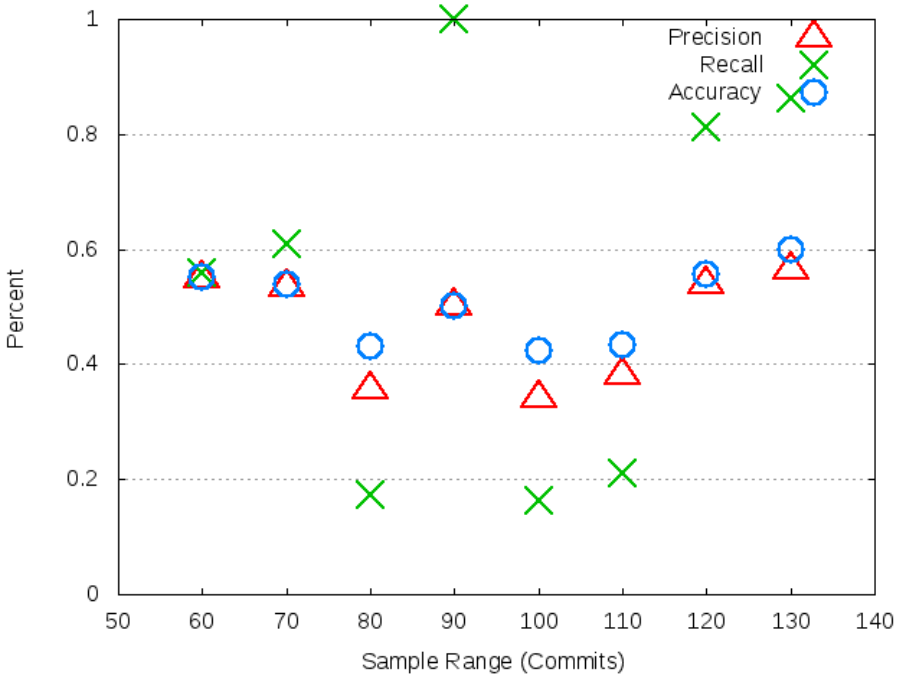


Figure 5.7: SWR for spark using SVM

Repository	AI	SWR	Precision	Recall	Accuracy
acra	SVM	100	0.74	0.92	0.8
arquillian-core	SVM	70	0.55	0.71	0.57
blockly-android	SVM	60	0.51	1.0	0.51
brave	SVM	90	0.52	0.58	0.52
cardslib	SVM	90	0.58	0.5	0.57
dagger	SVM	60	0.65	0.68	0.66
deeplearning4j	SVM	120	0.58	0.55	0.58
fresco	SVM	130	0.57	0.67	0.58
governator	SVM	110	0.62	0.5	0.6
greenDAO	SVM	60	0.5	1.0	0.5
http-request	SVM	80	0.59	0.93	0.65
ion	SVM	70	0.56	0.62	0.57
jadx	SVM	130	0.55	0.82	0.58
mapstruct	SVM	70	0.6	0.88	0.65
nettosphere	SVM	100	0.57	0.64	0.58
parceler	SVM	70	0.66	0.54	0.63
retrolambda	SVM	80	0.52	0.96	0.53
ShowcaseView	SVM	120	0.59	0.85	0.63
smile	SVM	100	0.58	0.67	0.6
spark	SVM	130	0.56	0.86	0.6
storm	SVM	100	0.51	0.7	0.52
tempto	SVM	120	0.66	0.5	0.62
yardstick	SVM	70	0.55	0.79	0.57

Table 5.9: SWR Repository Best Performance using SVM

Both smile in Figure 5.6 and spark in Figure 5.7 performed poorly each with performance measure below 0.5. In two cases (90 and 110) smile and 0 recall and undefined precision.

The best results for each repositories are outlined in Table 5.9. Overall there was no clear value for the SWR which held consistent positive results. Repositories from similar groups tended to perform similarly. For example acra, dagger and ShowcaseView all tended to perform well for similar parameters.

Repositories that were influenced more by SWR thus having a larger variation between values proved to have better results more often however this was not guar-

anteed. No value of SWR works across repositories and even for repositories that worked the correct value had to be found in order to obtain good results.

5.3.1.2 Feature Set Experiments

Extended Window	**Over Sampling**	**Under Sampling**	**Sample Rate**	**Window Offset**	**SWR**	**SVM C**	**gamma**
No	No	Yes	100%	5	90	10	8

Table 5.10: Feature Experiment Setup

This experiment uses different sets of candidate feature to test to explore the available features. The remaining variables were kept constant to allow for the candidate feature sets to be viewed in isolation. These constants are provided in Table 5.10. The value of 90 was selected for the SWR based on the value being in the middle of the range experimented on for the previous experiment. The remaining variables are kept the same as the previous experiment in Section 5.3.1.1.

Feature	Com	Sig	Name	f_Δ	sf_Δ	t_Δ	Length	$change_{t-1}$
1	•	•	•	•	•		•	•
2	•	•	•	•		•	•	•
3	•	•	•	•		•		•
4		•	•	•		•		•
5	•	•	•	•				•

Table 5.11: Candidate Feature Sets

The candidate feature sets are outlined in Table 5.11. These feature sets were selected from a larger set of features outlined in Section 4.1. Each set is assigned an index value to allow for easier reference later on. For the remainder of this section the experiment sets will be referenced using the assigned index. Therefore if feature set 3 is referenced then that refers to the candidate feature set in the third row. Some

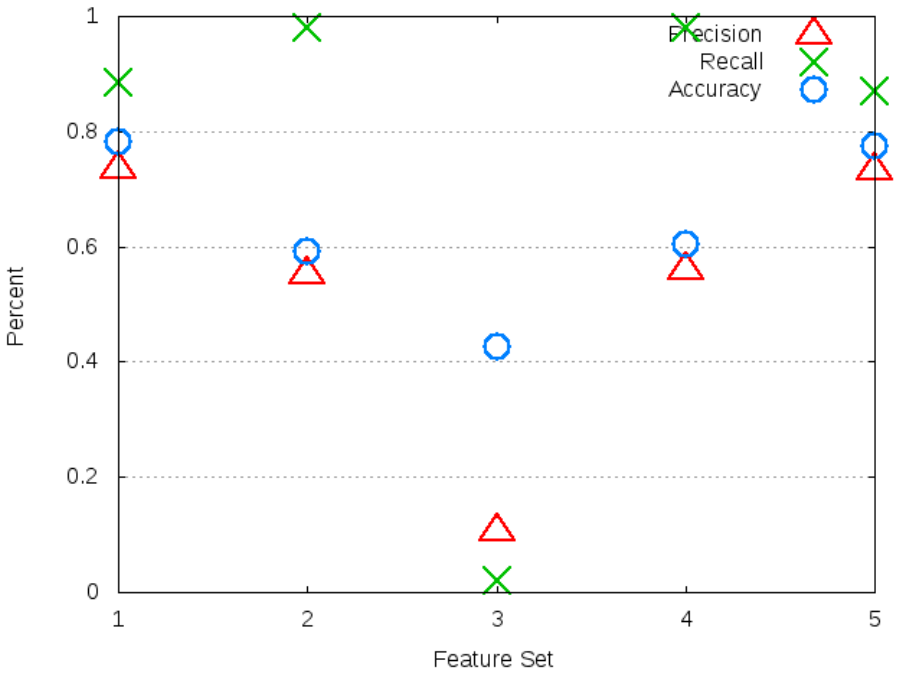


Figure 5.8: Feature for ShowcaseView using SVM

of the repositories results are shown in figures below. The rest of this experiments performance results can be found in subsection A.2.1.

The repositories ShowcaseView, deeplearning4j and ion were all greatly impacted by the different feature sets. ShowcaseView in Figure 5.8 performed well for feature set 1 and 5 and terribly for feature set 3. Similarly for ion in Figure 5.10, feature sets 1 and 5 performed well with the rest of the feature sets performing poorly. Finally for deeplearning4j in Figure 5.9, the best performance was for feature set 3 where as the remaining trails were not as good. There were a few repositories like these ones were one or two of the feature sets would perform well. One that performed well for certain feature sets tended to share similar repository classifications like ShowcaseView and ion do.

A lot of repositories did not vary greatly for different feature sets providing similar to results to that of nettosphere in Figure 5.11. All three performance measures

Repository	AI	Feature Set	Precision	Recall	Accuracy
acra	SVM	4	0.77	0.78	0.77
arquillian-core	SVM	4	0.55	0.83	0.57
blockly-android	SVM	2, 3, 4	0.5	1.0	0.5
brave	SVM	3, 4	0.6	0.51	0.58
cardslib	SVM	5	0.51	0.88	0.52
dagger	SVM	1	0.51	0.9	0.52
deeplearning4j	SVM	3	0.61	0.84	0.65
fresco	SVM	3	0.5	0.99	0.5
governator	SVM	1	0.65	0.57	0.63
greenDAO	SVM	3, 4	0.5	1.0	0.5
http-request	SVM	1	0.66	0.7	0.67
ion	SVM	5	0.58	0.83	0.61
jadx	SVM	3, 4	0.52	0.84	0.53
mapstruct	SVM	3	0.57	0.9	0.61
nettosphere	SVM	2	0.56	0.64	0.57
parceler	SVM	1	0.57	0.92	0.61
retrolambda	SVM	1	0.57	0.7	0.59
ShowcaseView	SVM	1	0.73	0.89	0.78
smile	SVM	5	0.53	0.96	0.55
spark	SVM	2, 4	0.5	1.0	0.5
storm	SVM	2	0.5	0.67	0.5
tempto	SVM	5	0.53	0.5	0.53
yardstick	SVM	3	0.53	0.68	0.54

Table 5.12: Feature Repository Best Performance using SVM

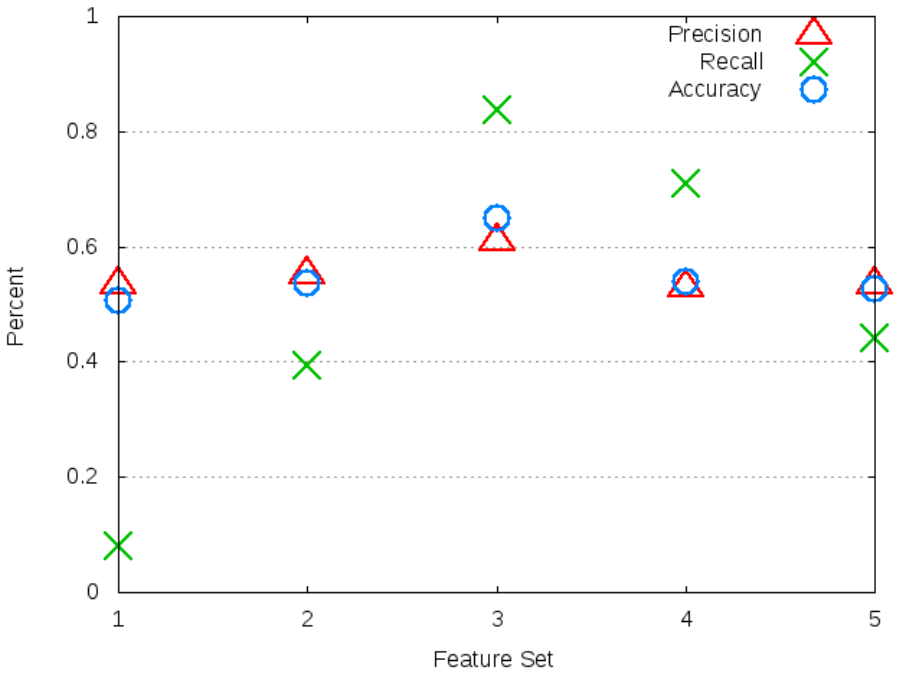


Figure 5.9: Feature for deeplearning4j using SVM

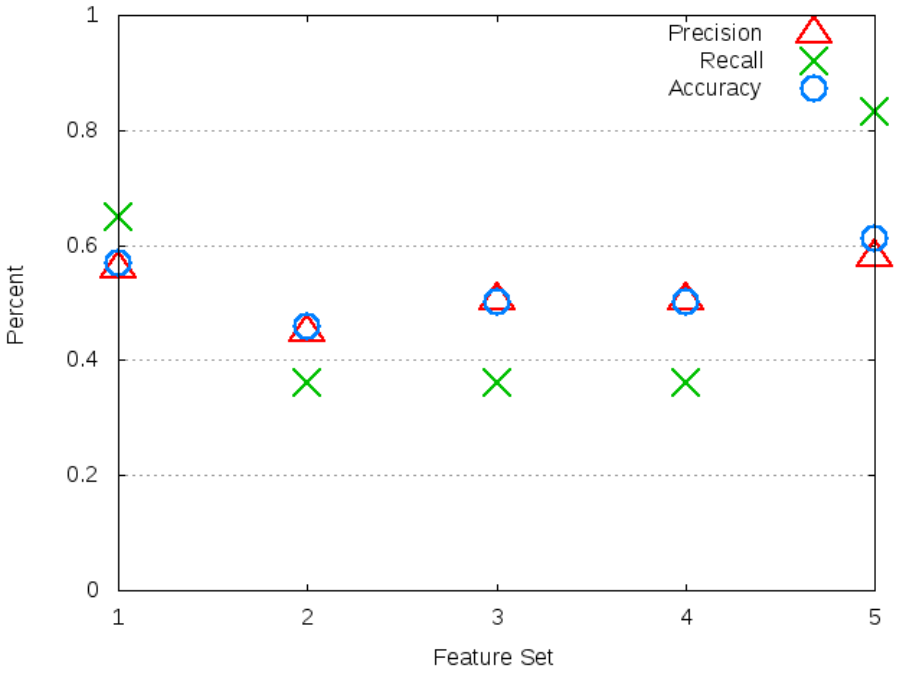


Figure 5.10: Feature for ion using SVM

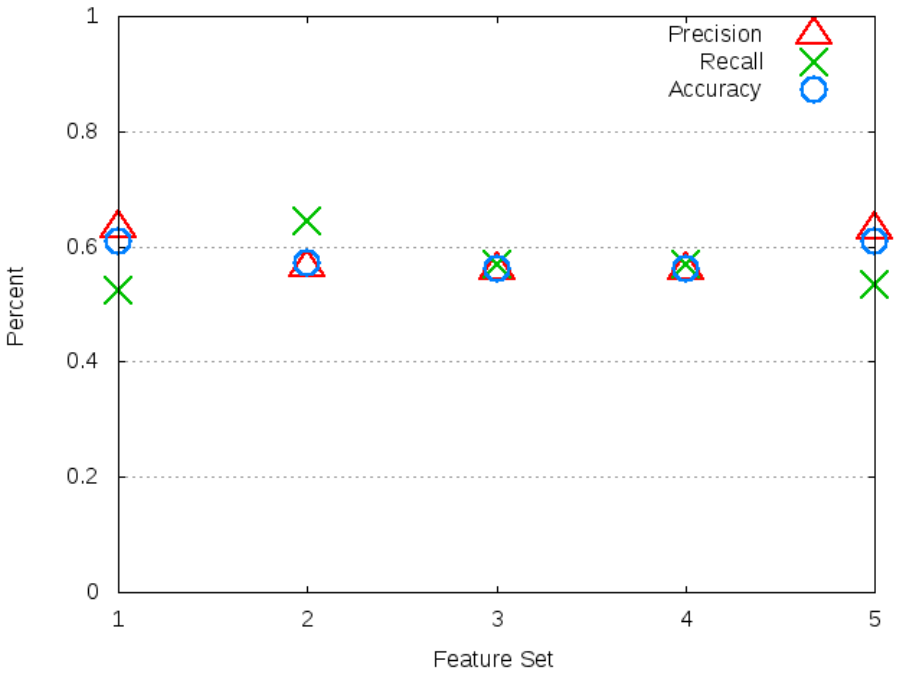


Figure 5.11: Feature for nettosphere using SVM

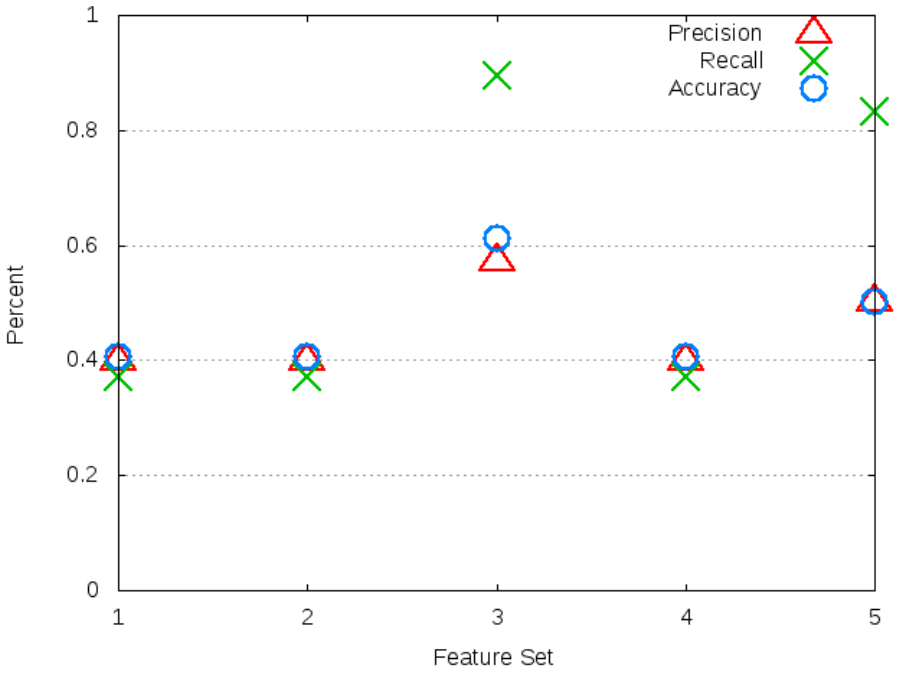


Figure 5.12: Feature for mapstruct using SVM

show little variance and only small chances are present between repositories. Other repositories performed poorly for all the feature sets such as mapstruct in Figure 5.12 which had 3 of the 5 trails score lower than 0.5 in all performance measures. In Table 5.12 each repository is outlined with the best performance for this experiment. The feature sets are shown to be an influencing factor on the performance of the model however no single feature set found to stand out as the ideal candidate for all repositories.

5.3.1.3 SVM Oversampling Experiment

Extended Window	Under Sampling	Sample Rate	Window Offset	SVM C	gamma
No	Yes	100%	5	10	8

Table 5.13: Feature Experiment Setup

Oversampling is a balancing technique used to increase the amount of samples available. Samples from the smaller data set are re-sampled to increase the size of the data set. While this does introduce duplicates into the model it also counter acts biasing that is present when one classification is more common then the other by a large margin. Under sampling is also used to remove excess elements from the larger set of classification. Oversampling This is especially useful for data sets that contain a small number of samples for a particular category. In that case under sampling may limit the performance of a model by removing nearly all of the elements in the data set.

The experiment below took the best and worst trials from the previous two experiments and used oversampling when sampling the data. The variables that change per repository are based on the previous best performance and worst performance.

Repository	Best		Worst	
	Feature Set	**SWR**	**Feature Set**	**SWR**
acra	2	80	3	90
arquillian-core	4	90	2	90
blockly-android	2	60	2	90
brave	2	130	3	90
cardslib	2	120	4	90
dagger	5	90	2	70
deeplearning4j	3	90	2	130
fresco	3	90	2	90
governator	1	90	3	90
greenDAO	4	90	1	90
http-request	2	80	3	90
ion	5	90	2	90
jadx	2	130	2	100
mapstruct	2	70	1	90
nettosphere	2	120	3	90
parceler	1	90	2	90
retrolambda	2	130	4	90
ShowcaseView	1	90	2	80
smile	2	70	2	90
spark	4	90	3	90
storm	2	100	2	110
tempto	2	120	2	130
yardstick	2	70	2	100

Table 5.14: Best And Worst Results From experiments 1 and 2 for SVM

In Table 5.14, the best and worst SWR and feature set are provided for each repository. Since each repository will likely have different values of SWR and feature set the comparesion should only be made between the difference in performance for the best/worst result and their corresponding oversampling trail Best-O/Worst-O.

In some cases the best performing experiment may not have been entirely clear. For example with some repositories having very high recall (≥ 0.9) while having lower precision and accuracy. The best trail was picked based on having the all a weighted summation algorithm outlined in Equation 5.9. Since precision and accuracy are very

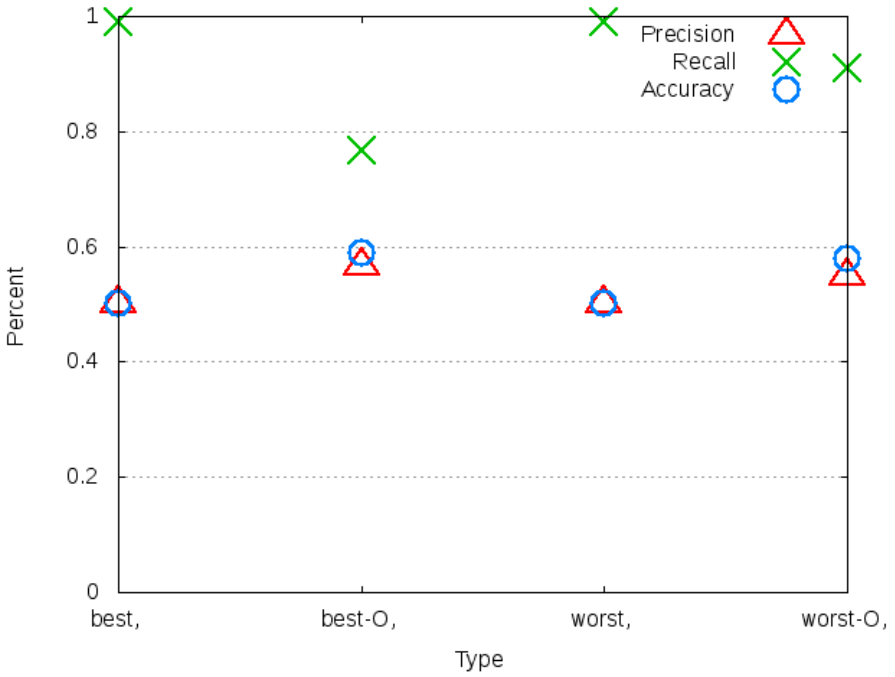


Figure 5.13: Oversampling for fresco using SVM

closely related the weight for each was 0.5 while recall was set to 1.0.

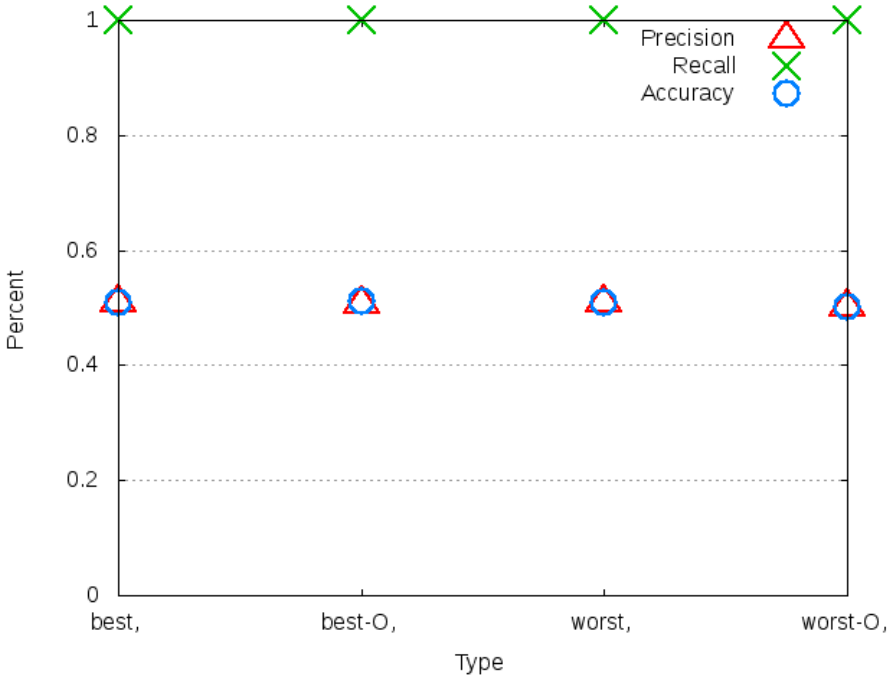


Figure 5.14: Oversampling for blockly-android using SVM

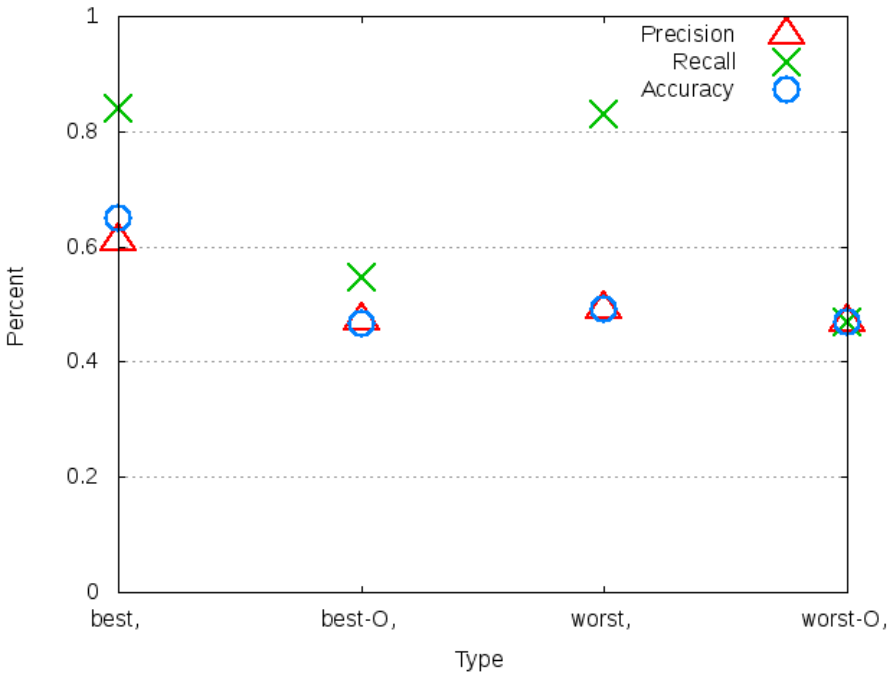


Figure 5.15: Oversampling for deeplearning4j using SVM

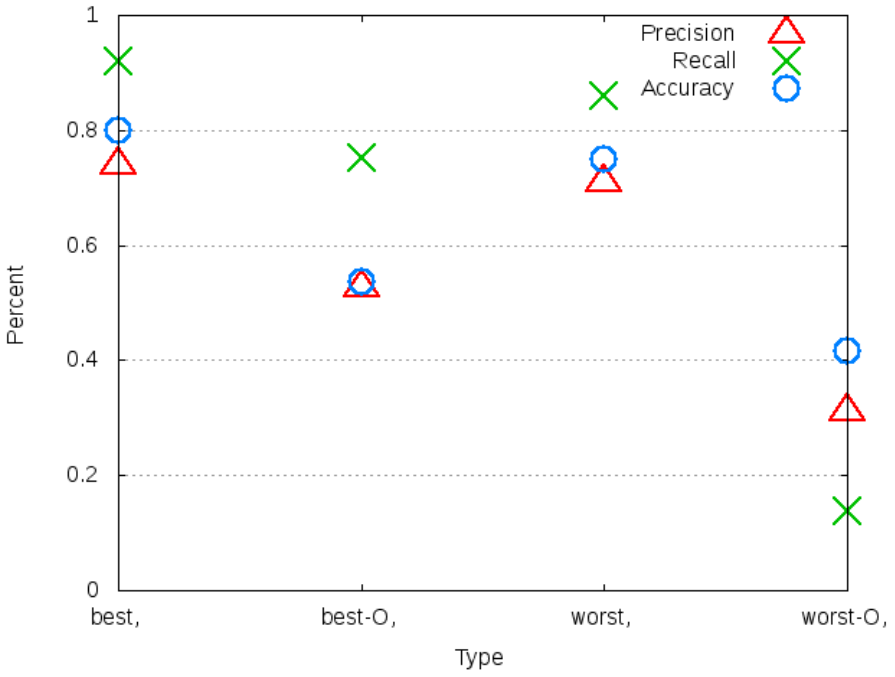


Figure 5.16: Oversampling for acra using SVM

Some repositories such as fresco Figure 5.13 performed a little better for precision and accuracy while a little worse for precision. Unlike most repositories however blockly-android in Figure 5.14 experienced very little change with the introduction of oversampling. Finally, the majority of the experiments showed oversampling to provide a negative impact of the performance of the model. Both deeplearning4j in Figure 5.15 and acra in Figure 5.16 performed worse for all measures for both trails.

The overall impact of using oversampling for training the model proved detrimental with the vast majority of repositories performing worse with the use of oversampling. While some repositories experience increases in individual performance measures, other measures fall. Also, any performance seen by a repository is minimal at best while the loss of performance tends to be substantial.

5.3.1.4 SVM Discussion

The three different experiments attempted to determine the impact of the different factors on the prediction method. The three factors that were tested are:

1. SWR
2. Model features
3. Sampling balancing

The results of the repositories did not follow any common trends between repositories. Similar repository groups outlined in Table 5.2 do not perform similarly for the most part with the exception of the group for acra, dagger and ShowcaseView which all had an okay but for different values of SWR. The only similarity that can be found is for groups of repository that performed poorly which still was inconsistent. For most repositories at the very least the recall was variable. The only really exception would be blockly-android which experienced little to no variation in for each trial.

Some repositories saw more variation in the precision and accuracy and often had at least trail which performed moderately well.

For the second experiment, the performance results were a lot lower than the first generally. This appears to be directly related to the impact that each of these variables has on the performance of the model. Therefore the focus is placed on the best feature set for all repositories or for groups of repositories. For some of the groups of repositories, a certain feature set or pair of feature sets performed well for all repositories in the group. For example, acra, dagger and ShowcaseView all performed well when using feature set 1 or 5. Not all of the repositories performed their best using that those feature sets. However, they did perform close to their best performance. While this trend occurred for some repositories it was not consistent for all repositories. Similarly there was no best performing feature set for all repository.

Finally of the third experiment was found to have a variable impact on the repositories when sample balancing through oversampling was applied. In terms of performance impact, oversampling provide a negative impact on most of the repositories experimented on. Some trials saw now change and a very small number of trails saw slight improvements to one performance measure while decreasing another performance measure. The slight improvements found from using oversampling were insignificant compared to the drop in performance typically experienced.

Overall SWR had the greatest impact on the performance of the prediction method. The model feature set had less of an impact and balancing the sample through oversampling provided a primarily negative impact. The SWR while having a larger impact on most repositories, some repositories were less affected. Generally variations of SWR could produce a positive results, a negative results were also present. Furthermore, no clear pattern was discovered to allow for simple configuration of the parameters to provide positive results. Therefore use of the approach with a SVM

model can be beneficial but also incurs a risk associated with poor predictions.

5.3.2 Random Forest Experiments

The machine learning algorithm RF is used for the second set of experiments. RF was selected as an alternative to SVM for it's success in various data mining related tools. The implementation of RF is in a python library *scikit-learn* which is outlined in Section 4.2. Only one parameter is used for RF, the forest size, which is set to 10000 all of these experiments.

5.3.2.1 Window Range Experiments

Com	Sig	Name	f_Δ	sf_Δ	t_Δ	Length	$change_{t-1}$
•	•	•	•		•	•	•

Table 5.15: SWR Experiment Features

Extended Window	**Over Sampling**	**Under Sampling**	**Sample Rate**	**Window Offset**	**RF Size**
No	No	Yes	100%	5	10000

Table 5.16: SWR Experiment Setup

The independent variable for this set of experiments is the sample window size measured in commits. The feature set are outlined in Table 5.15. The features used for this experiment is the same as the first SVM experiment feature set.

The parameters for this experiment are outlined in Table 5.16. The only difference between the parameters used in this experiment and the parameters used in the SVM experiment one is the RF specific parameters. This allows for a fairly clear comparison between these two methods with the given independent variable, the

SWR. The experiment was conducted on all 23 repositories collected and examples were are discussed in more detail in this section. The remainder of the repositories performance results are outlined in subsection A.1.2.

Each repositories experimental results are accompanied with a second figure that outlines the importance for each feature set variable in the creation of the prediction model. The importance of a feature only identifies how influential that feature was to the prediction and therefore necessitates the context of the results. So if a repository performs poorly in predicting the most influential features are more likely to not as useful for predictions with the repository. Likewise, if a repository performs well the corresponding feature importance can indicate highly influential features that helped produce positive results.

Even in the case of where a repository performs well the feature set used may not be generalizable to other repositories. For example http-request in Figure 5.18 performs well with a SWR of 70 - 90 and places high importance on Sig, $Name$, $time$ and $length$. Alternatively, dagger in Figure 5.20 performs moderately well with an SWR of 60 and 100-110 while placing a higher importance on Com and $time$. Even more interesting is that http-request placed nearly 0 importance on Com while the same feature ranked second for dagger.

The performance of each repository varied, with a few repositories performing well for some SWR like http-request in Figure 5.17, dagger in Figure 5.19 and ShowcaseView in Figure 5.21. The impact of the changes to SWR is clearly visible as some trails perform poorly, while others perform a lot better. For example in ShowcaseView, use of a SWR of 100 or higher provides good performance but below the performance is a lot lower. For some repositories such as jadx in Figure 5.23 the SWR had less of an impact causing little variation between precision and accuracy while offering only slight changes with the recall.

Repository	AI	SWR	Precision	Recall	Accuracy
acra	RF	90	0.78	0.75	0.77
arquillian-core	RF	90	0.52	0.98	0.54
blockly-android	RF	70	0.59	0.65	0.6
brave	RF	90	0.6	0.95	0.65
cardslib	RF	100	0.65	0.74	0.67
dagger	RF	110	0.72	0.69	0.71
deeplearning4j	RF	120	0.57	0.93	0.61
fresco	RF	60	0.52	1.0	0.53
governator	RF	60	0.57	0.81	0.6
greenDAO	RF	120	0.54	0.54	0.54
http-request	RF	80	0.87	0.79	0.84
ion	RF	90	0.71	0.73	0.72
jadx	RF	130	0.55	0.75	0.56
mapstruct	RF	110	0.58	0.88	0.62
nettosphere	RF	110	0.63	0.67	0.63
parceler	RF	100	0.77	0.63	0.72
retrolambda	RF	70	0.56	0.73	0.58
ShowcaseView	RF	130	0.78	0.83	0.8
smile	RF	100	0.53	0.84	0.54
spark	RF	110	0.54	0.91	0.57
storm	RF	60	0.61	0.89	0.66
tempto	RF	70	0.58	0.65	0.59
yardstick	RF	70	0.54	0.67	0.55

Table 5.17: SWR Repository Best Performance using RF

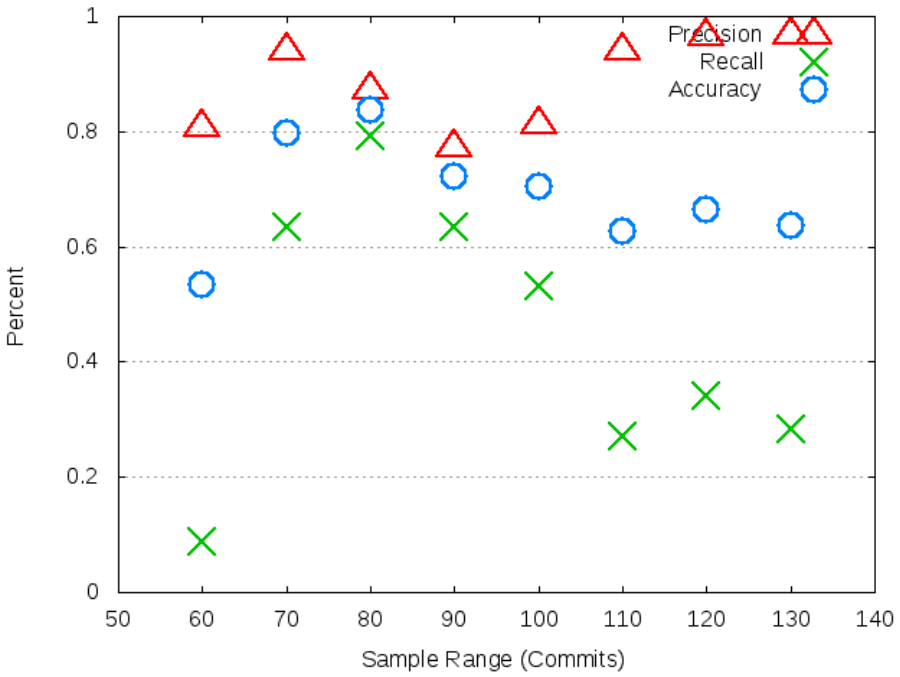


Figure 5.17: SWR for http-request using RF

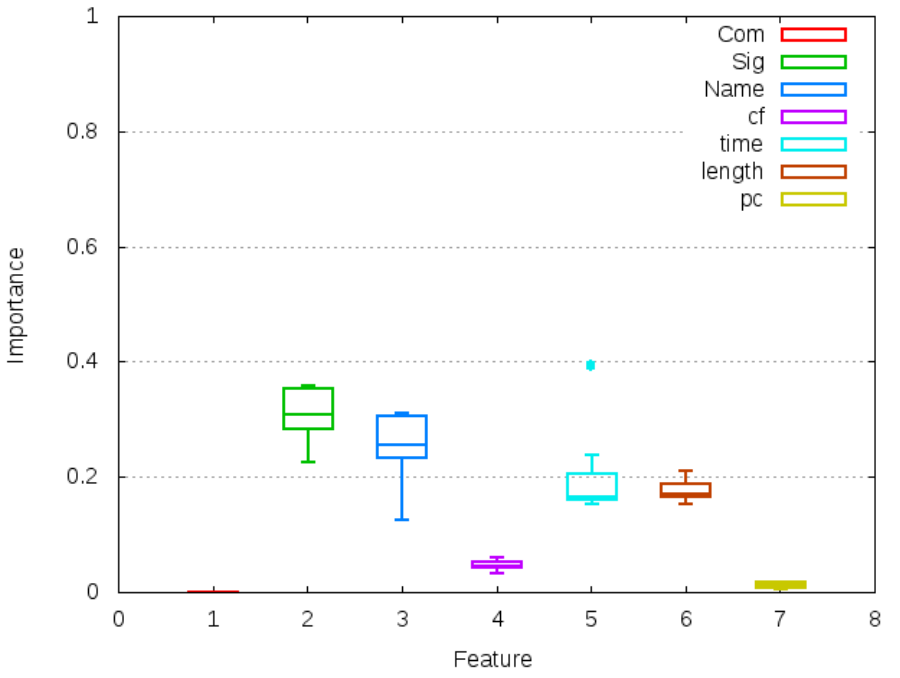


Figure 5.18: Feature Importance SWR for http-request using RF

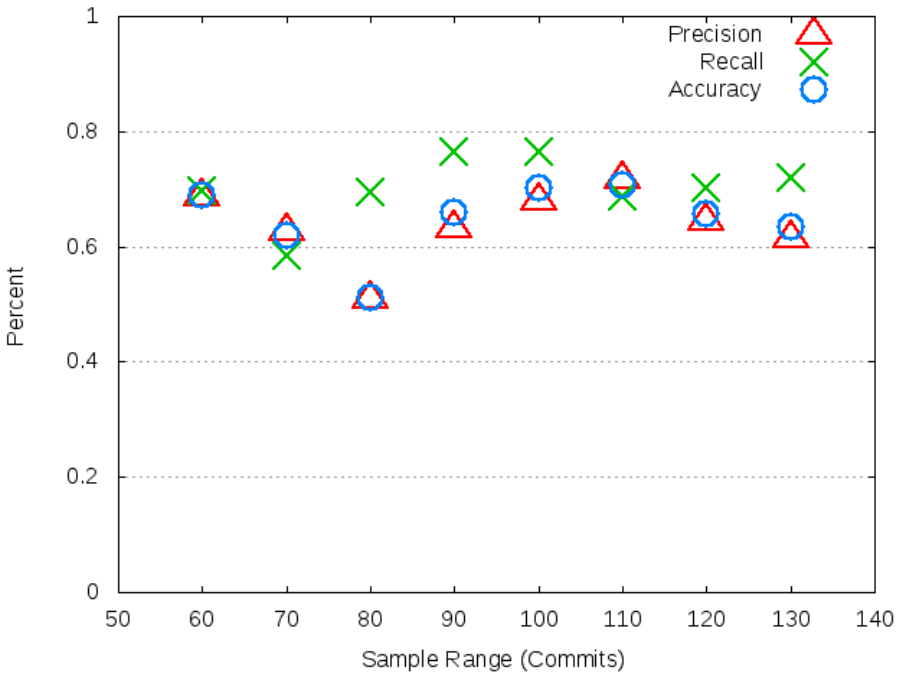


Figure 5.19: SWR for dagger using RF

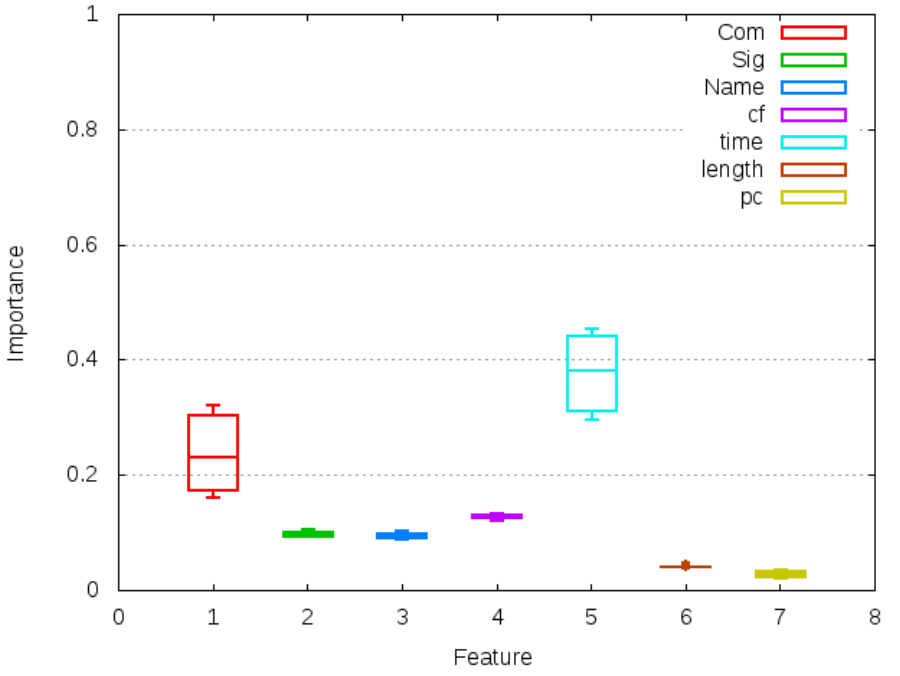


Figure 5.20: Feature Importance SWR for dagger using RF

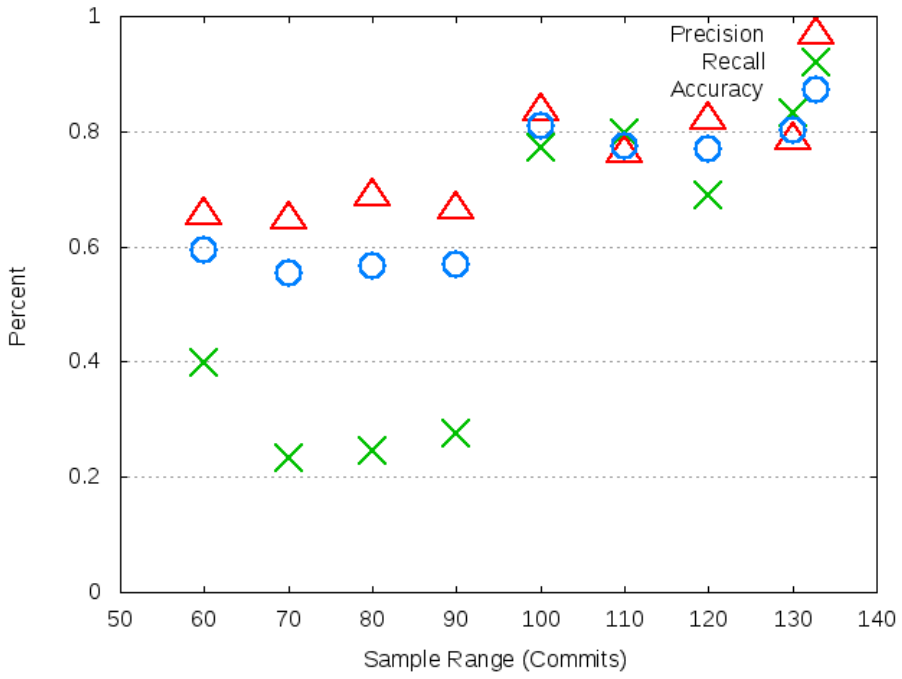


Figure 5.21: SWR for ShowcaseView using RF

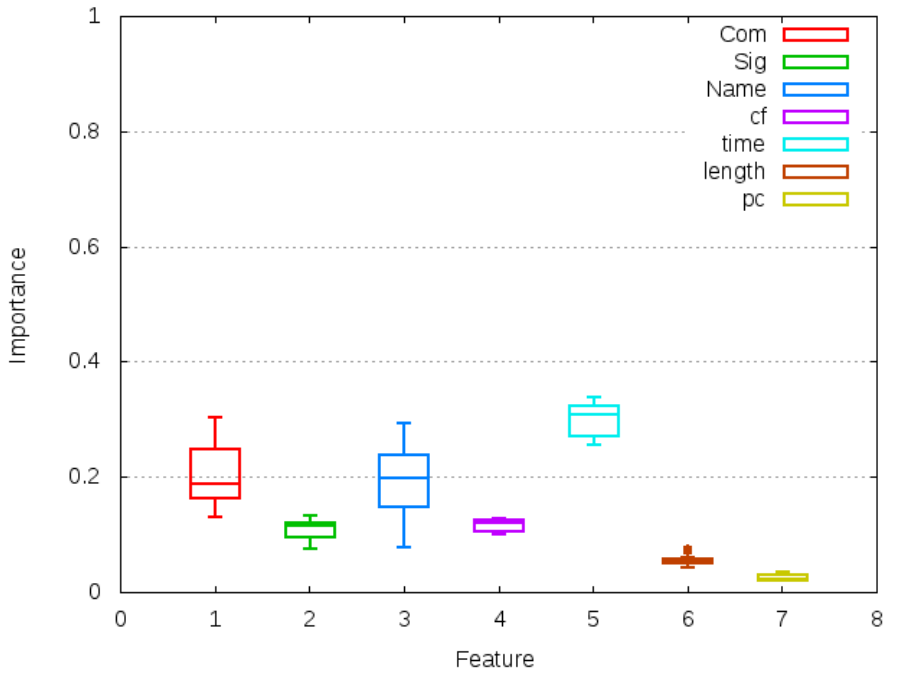


Figure 5.22: Feature Importance SWR for ShowcaseView using RF

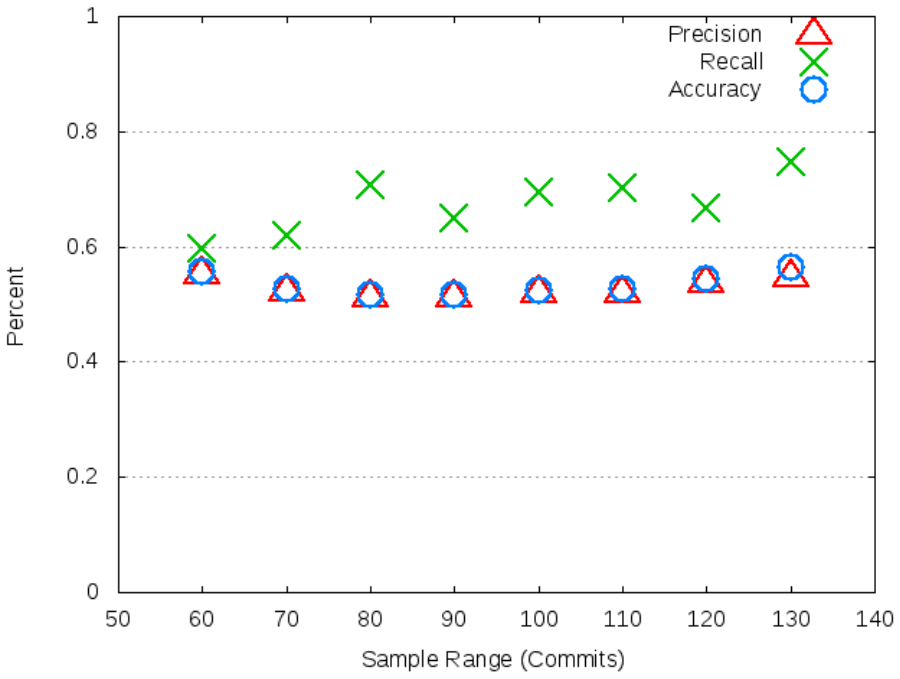


Figure 5.23: SWR for jadx using RF

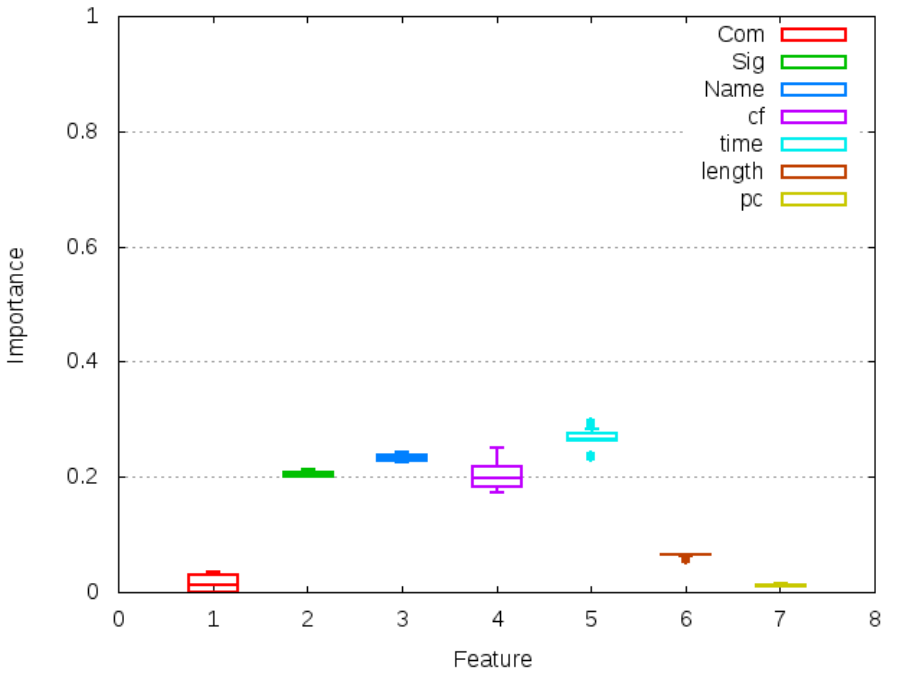


Figure 5.24: Feature Importance SWR for jadx using RF

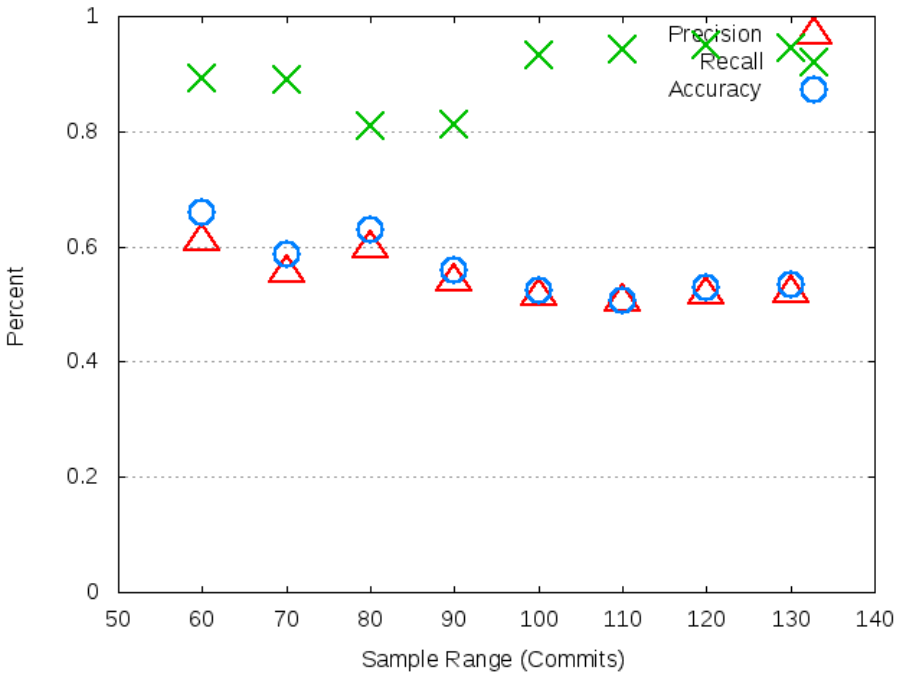


Figure 5.25: SWR for storm using RF

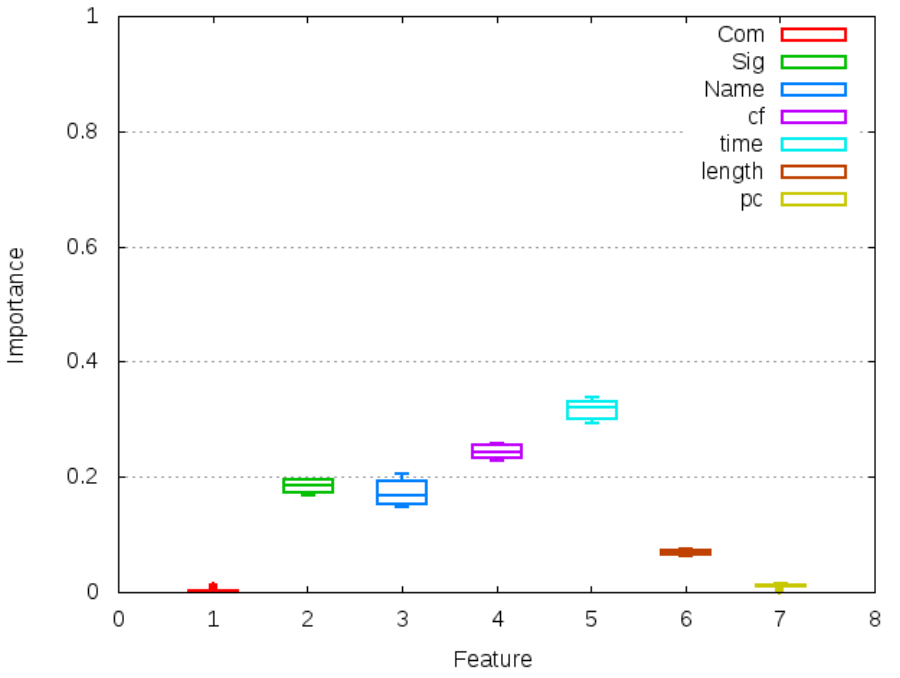


Figure 5.26: Feature Importance SWR for storm using RF

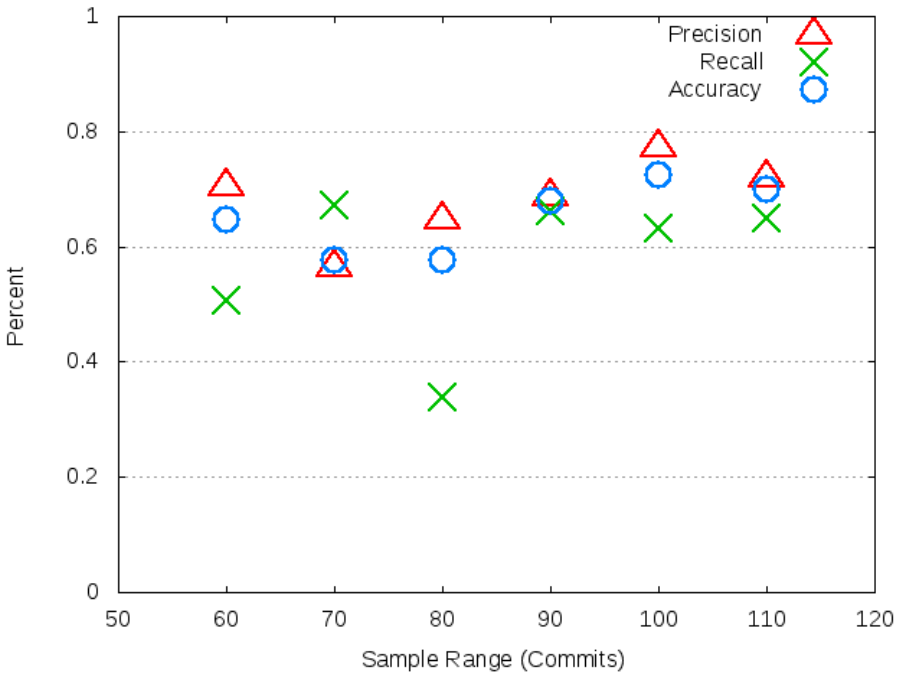


Figure 5.27: SWR for parceler using RF

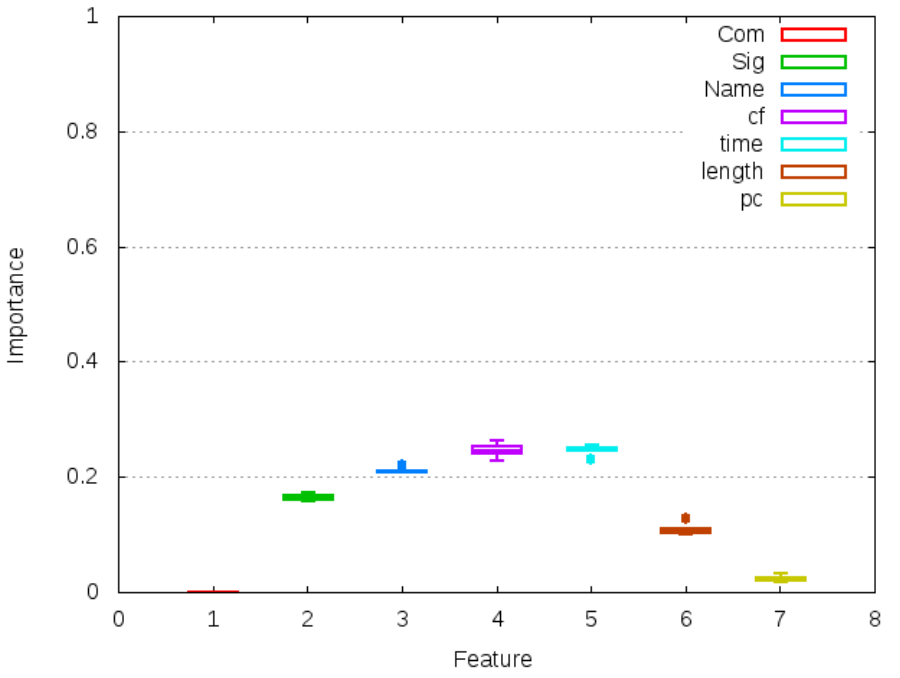


Figure 5.28: Feature Importance SWR for parceler using RF

In Table 5.17, the best performance is shown for each repository for this experiment. Some repositories experienced very little impact from variations of the SWR for the performance. For example, storm in Figure 5.25 had higher recall but lower precision and accuracy. Other factors may provide more influence such smaller or larger values of SWR however those were outside the scope of the experiment. Finally, other repositories did not perform as well but experienced some variation to precision, recall and accuracy. One such repository would be Figure 5.27 which had all three measures fairly close together for most trails but because of the size of the repository could not supply sufficient data for a SWR of 120 or 130. The importance of both storm and parceler was similar to that of http-request however as noted neither managed to perform as well as the best performance from http-request.

5.3.2.2 Feature Set Experiments

Extended Window	**Over Sampling**	**Under Sampling**	**Sample Rate**	**Window Offset**	**SWR**	**RF Size**
No	No	Yes	100%	5	90	10000

Table 5.18: Candidate Feature Experiment Setup

Feature	Com	Sig	Name	f_Δ	sf_Δ	t_Δ	Length	$change_{t-1}$
1	•	•	•	•	•		•	•
2	•	•	•	•		•	•	•
3	•	•	•	•		•		•
4		•	•	•		•		•
5	•	•	•	•				•

Table 5.19: Candidate Feature Sets

Similar to the experiment using a SVM in Section 5.3.1.2. The experiment parameters are outlined in Table 5.18. The candidate features are likewise outlined in

Table 5.19. Each set is assigned an index value to allow for easier reference later on in this section. The candidate feature set will be referenced by the index assigned in the plots and discussions related. The candidate feature sets were used experimented on with each repository which are discussed below. The following results are highlights of the larger set of experiments conducted on each repository. The remainder of the results for this experiment are outlined in subsection A.2.2.

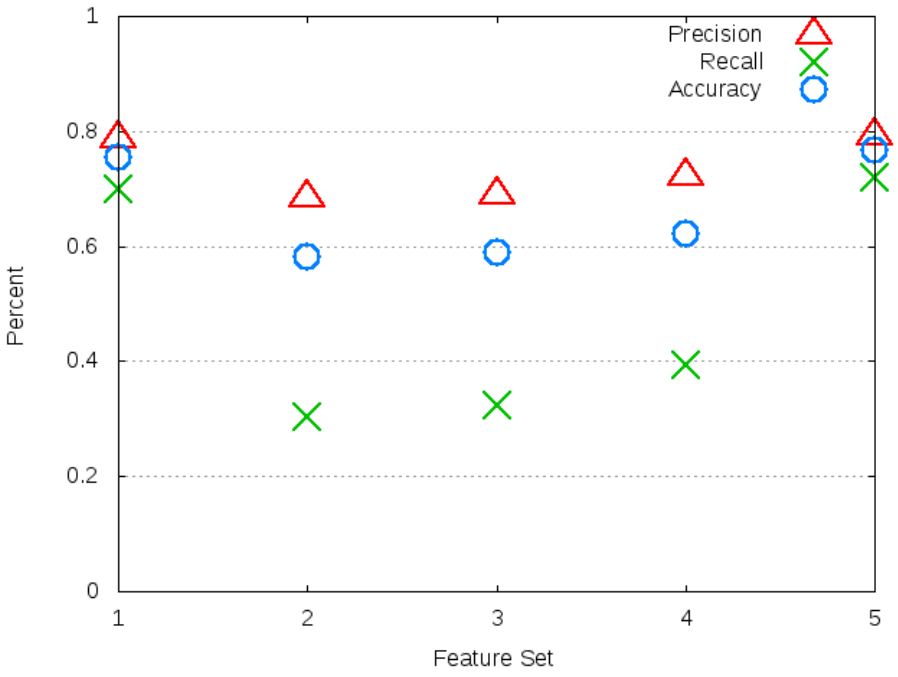


Figure 5.29: Feature for ShowcaseView using RF

The repository to perform the best in this experiment was ShowcaseView in Figure 5.29 which performed best for feature sets 1 and 5 and had minimal difference between the performance of feature sets 2, 3 and 4. Typically most repositories performed well with two feature sets or more which most likely is related to the similarity in the feature sets tested. A few of the successful repositories, such as ion in Figure 5.30, performed well consistently for each feature set. Finally, other more successful repositories saw dramatic variations between the different feature sets. For

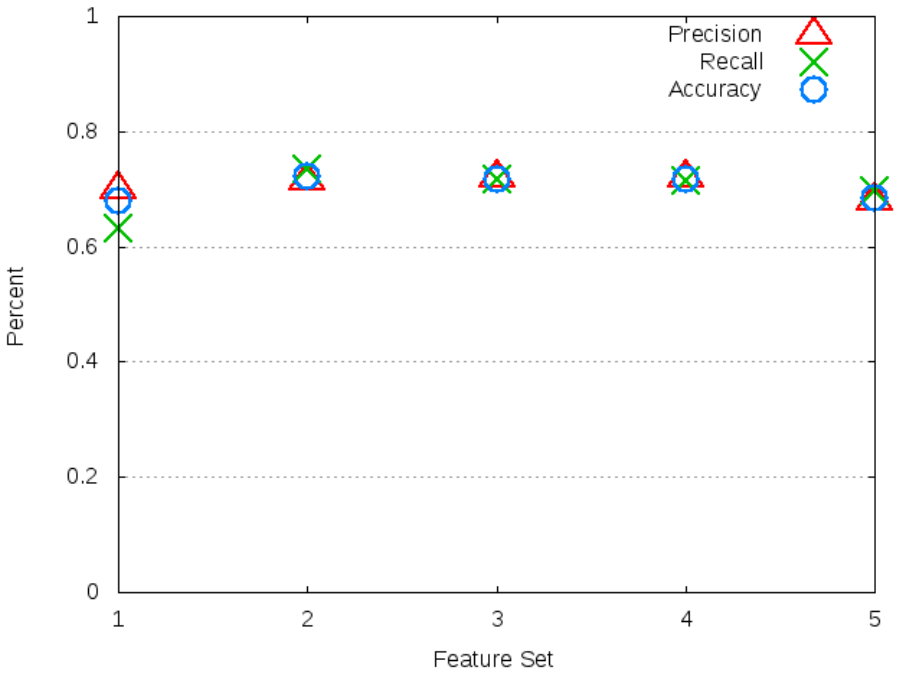


Figure 5.30: Feature for ion using RF

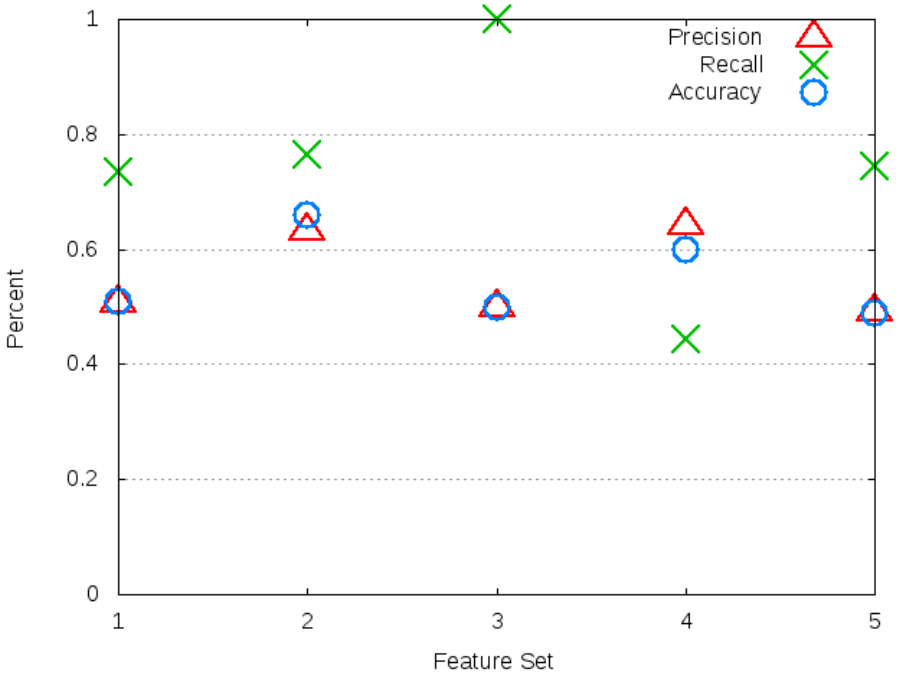


Figure 5.31: Feature for dagger using RF

example, dagger in Figure 5.31, performs well in the second feature set and poorly in the rest. The recall is especially volatile at dropping below 0.5 for feature set 4.

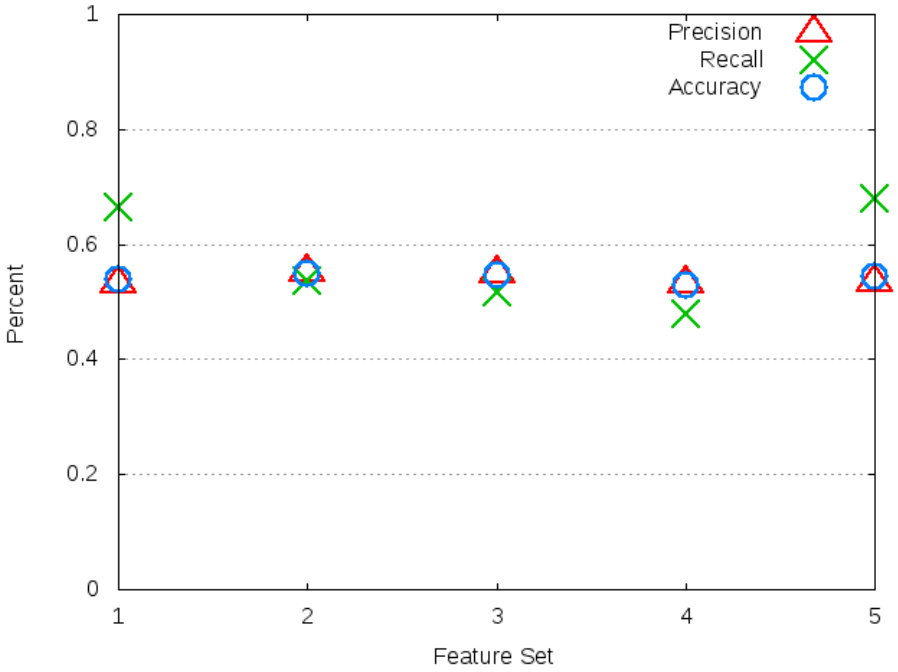


Figure 5.32: Feature for cardslib using RF

The majority of the repositories experimented on saw little difference for each feature set. The best performance for each repository is outlined in Table 5.20. For cardslib in Figure A.97, the precision and accuracy stay right above 0.5 while the recall dips below 0.5 for feature set 4. The performance is not great and overall the impact of the different feature sets appears quite low for this repository. Alternatively, governator in Figure A.101, is one of the few repositories to perform well for precision while low for accuracy and very low for recall. Each feature set again has only a small impact on the performance with the repository performing very poorly for every trial. Overall the results for this experiment were mixed since the impact of the feature set at least of these three features is less significant that the SWR.

Repository	AI	Feature Set	Precision	Recall	Accuracy
acra	RF	4	0.77	0.78	0.77
arquillian-core	RF	5	0.53	0.97	0.56
blockly-android	RF	2	0.56	0.73	0.58
brave	RF	2	0.53	0.92	0.55
cardslib	RF	5	0.53	0.68	0.54
dagger	RF	2	0.63	0.76	0.66
deeplearning4j	RF	3	0.54	0.91	0.56
fresco	RF	3	0.5	1.0	0.5
governator	RF	3	0.83	0.25	0.6
greenDAO	RF	3	0.52	0.47	0.52
http-request	RF	1	0.67	0.7	0.68
ion	RF	2	0.72	0.73	0.72
jadx	RF	3	0.51	0.67	0.52
mapstruct	RF	1	0.54	0.95	0.57
nettosphere	RF	5	0.76	0.42	0.64
parceler	RF	3	0.68	0.7	0.69
retrolambda	RF	1	0.59	0.17	0.53
ShowcaseView	RF	5	0.79	0.72	0.77
smile	RF	4	0.51	0.85	0.52
spark	RF	1	0.52	0.72	0.52
storm	RF	5	0.54	0.85	0.57
tempto	RF	4	0.51	0.67	0.52
yardstick	RF	2	0.62	0.24	0.55

Table 5.20: Feature Repository Best Performance using RF

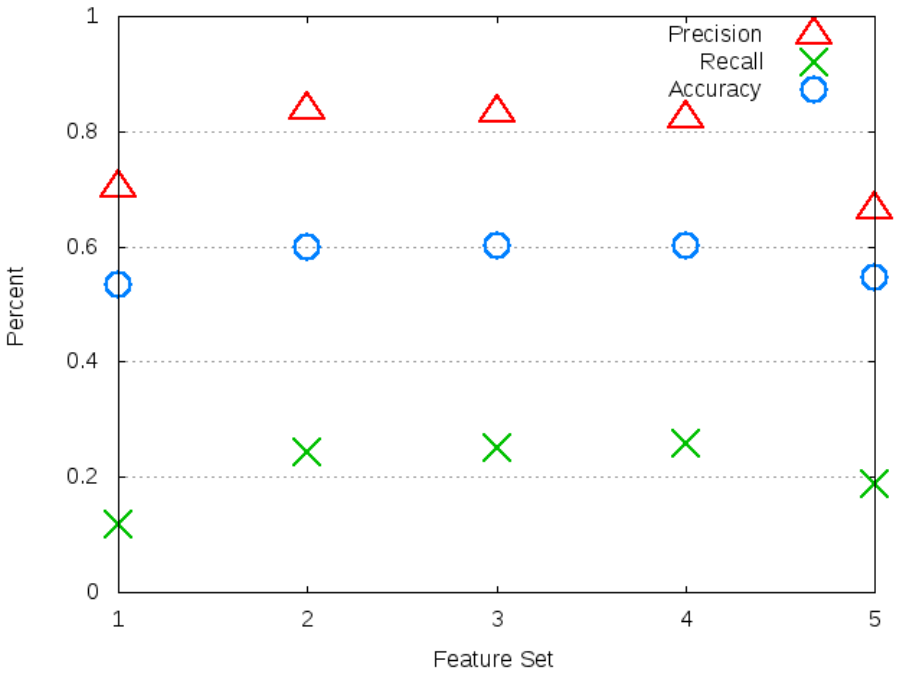


Figure 5.33: Feature for governator using RF

5.3.2.3 Oversampling Experiment

Extended Window	Under Sampling	Sample Rate	Window Offset	RF Size
No	Yes	100%	5	10000

Table 5.21: Oversampling Experiment Setup

This experiment builds on top of the previous two experiments and shares a very similar setup to those experiments.The experiment parameters are outlined in Table 5.21. The best and worst trails for each repository are taken from the previous two experiments. The value for SWR and the feature set used were recorded in Table 5.22 for each repository for the best and worst performance of the RF model. The experiment applies oversampling the best and worst trials to compare the performance of the model with and without the use of oversampling.

The result of a trail with oversampling are represented in the figures by either

Repository	Best		Worst	
	Feature Set	**SWR**	**Feature Set**	**SWR**
acra	2	60	5	90
arquillian-core	3	90	4	90
blockly-android	2	90	1	90
brave	2	110	4	90
cardslib	2	100	4	90
dagger	3	90	2	80
deeplearning4j	2	70	2	80
fresco	2	60	2	90
governator	2	60	2	90
greenDAO	2	60	2	100
http-request	2	80	4	90
ion	2	90	1	90
jadx	2	130	2	70
mapstruct	1	90	5	90
nettosphere	2	110	2	90
parceler	3	90	1	90
retrolambda	2	120	2	90
ShowcaseView	2	130	2	90
smile	5	90	2	100
spark	2	110	2	80
storm	2	60	2	90
tempto	2	120	2	130
yardstick	2	70	2	60

Table 5.22: Best And Worst Results From Experiments 1 and 2 for RF

best-O or *worst-O*. The results without oversampling from the previous experiment are represented with *best* and *worst*.

The results for the use of oversampling on the best and worst trails from each repository provide to have little effect on the results. In rare cases such as dagger in Figure 5.34, the performance marginally improved for the some measures while decreasing for others. However dagger also performed worse when oversampling was used on the worst trial. Similarly yardstick in Figure 5.35, performed slightly better for best-O and dramatically worse for worst-O.

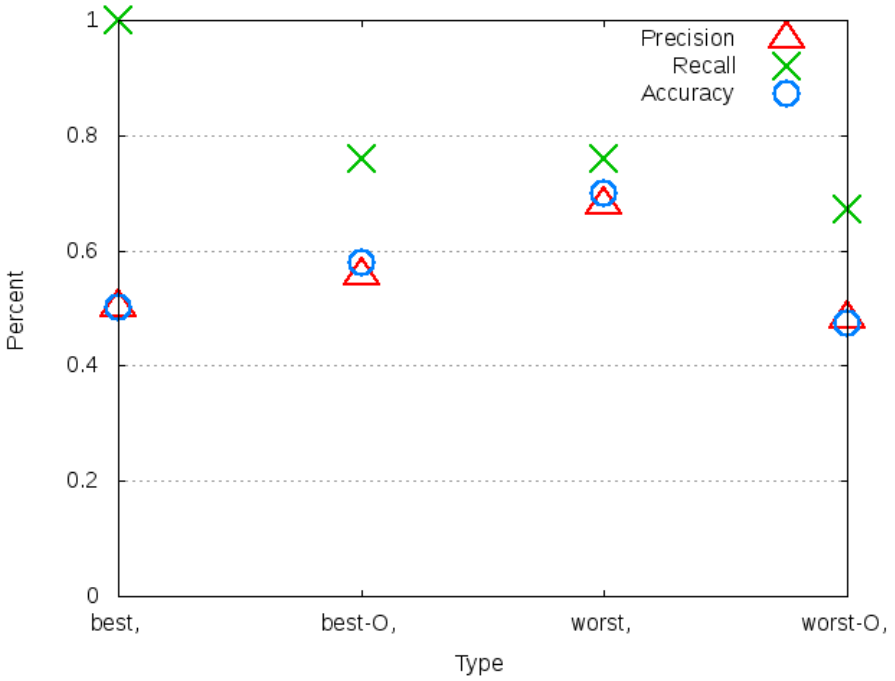


Figure 5.34: Oversampling for dagger using RF

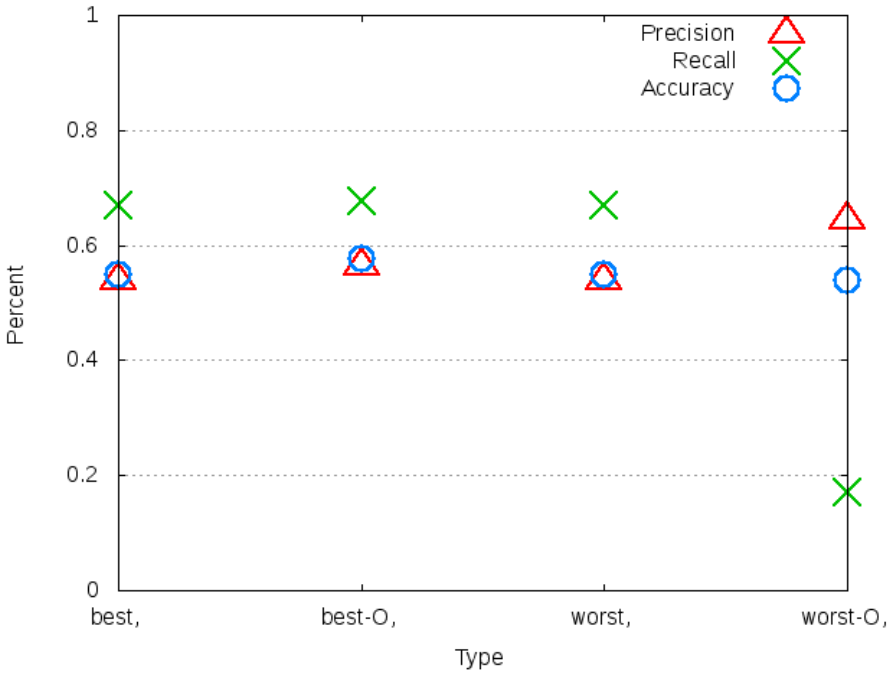


Figure 5.35: Oversampling for yardstick using RF

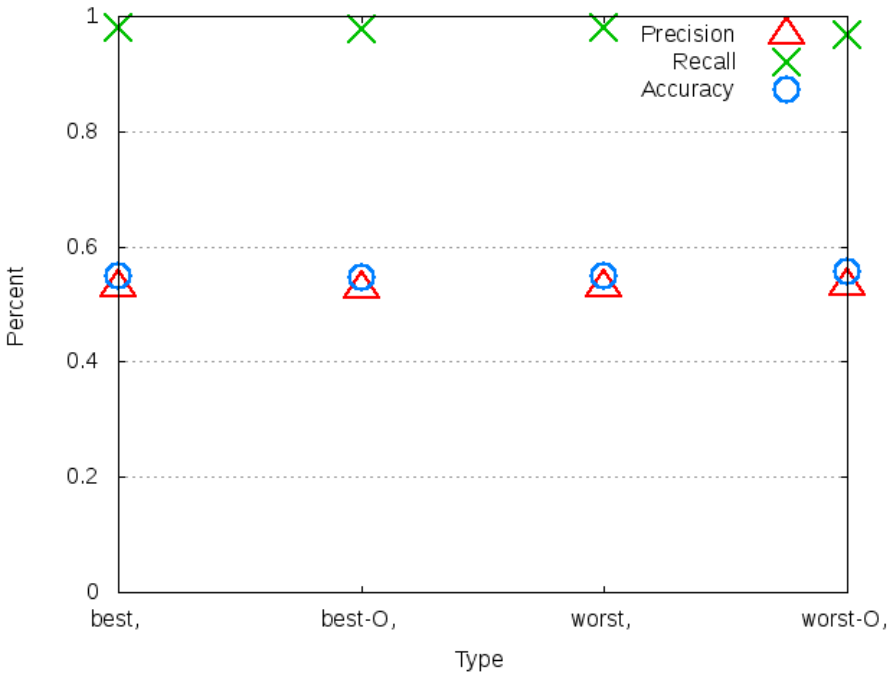


Figure 5.36: Oversampling for arquillian-core using RF

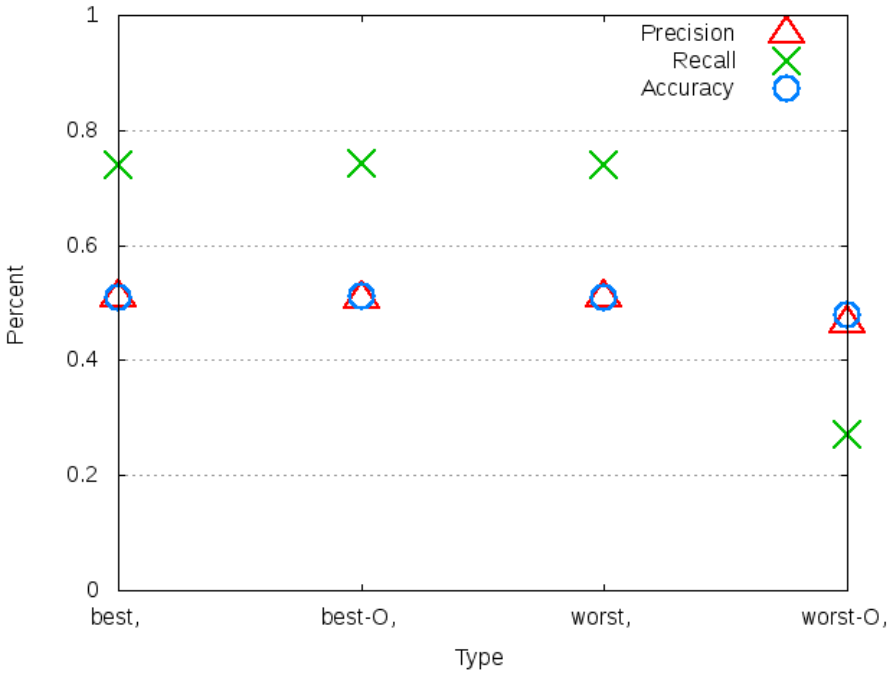


Figure 5.37: Oversampling for greenDAO using RF

The majority of the repositories however saw no improvement or a slight decrease in performance. For example arquillian-core in Figure 5.36 performed around the same for best-O and slightly worse for worst-O. Generally the differences if any were very small for most. Finally, some repositories performed a lot worse for their worst-O trial including greenDAO in Figure 5.37. Overall the use of oversampling on either the best or worst case cause little difference. The only major difference was for worst-O which typically performed poor in compared to the trial without the use of oversampling.

5.3.2.4 Random Forest Discussion

The approach was experimented on using the machine learning algorithm RF. The three factors; SWR, feature set and oversampling were investigated. The first factor, SWR, achieved positive results for some of the repositories and appeared to have the largest impact on the performance. While the impact of the SWR was larger often the best result for a repository would of accuracy and precision at 0.5 and recall at 1.0. The impact of the feature set should also not be overshadowed as some repositories performed better in the third experiment than the first. Finally, the impact of sample balancing on the data set proved primarily negative with both the best and worst trails performing worse when oversampling was used.

The first experimental results did not show a pattern for present for all the repositories. Some repositories performed better while a lot performed mediocre at just above 0.5 precision, accuracy. A few of the repositories had poor performances with measures dropping below 0.5. Most of the groups yielded positive similar results, with acra, dagger and ShowcaseView all performing well and having fairly large variations for different trials. Likewise, aqruillian-core, brave and spark all performed similarly. While the results for these three repository were not particularly good they

still showed similar trends of performance.

For the second experiment no particular feature set was found to work for all repositories. Some repositories performed well for some while others performed well for others. Generally though, repositories tended not perform well for all feature sets. One repository such repository to perform poorly was governator which performed well for precision, okay for accuracy and terribly for recall. However in terms of repository groupings, the grouping of acra, dagger and ShowcaseView all performed well but varied in which feature set they performed well with. Overall groups tended to perform differently and had success with different feature sets. A feature set that worked well for one repository in the group could just as easily work poorly for another. This is highlighted best in the group consisting of http-request, nettosphere, parceler and retrolamdba. Both http-request and parceler perform well and nettosphere and retrolamdba perform for badly all feature sets.

Of course the final experiment with oversampling performed similar to the SVM experiments. For most repositories the use of oversampling in balancing the sample data provided no difference in model performance. The second most common outcome was for trails that used oversampling to result in lower performance than without. Finally in some rare cases, 2 of the 23 repositories saw a one of their trials perform slightly better with the use of oversampling. The results were not consistent for these repositories since the first repository was dagger saw the other corresponding trail decrease in performance. However the second repository, smile, recorded no difference in result for the corresponding trail.

Overall for the experiments using RF, the variable with the greatest impact was the SWR. However these variables were not consistent across repositories. Even for repositories that performed well the variable values differed. Repositories that worked tended to be repositories that were long, small in size, with a small to medium number

of developers and a low to medium rate of commits. This outcome is similar to the experiments conducted with SVM.

5.3.3 Experiment Discussions

The best performance for each repository with respect to predictive method and factors are out of the experiments conducted are shown in Table 5.23. Therefore the best parameters are outlined for each repository. The best result was calculated through multi-variable optimization. Since the performance measure consisted of three variables the best result would be one that maintained higher results for each. While not the most ideal, a weighted sum was taken of the precision, recall and accuracy outlined in Equation 5.9. The weight of precision (w_p) and accuracy (w_a) are closely related both were assigned a weight of 0.25 while the weight for recall (w_r) was assigned 0.5. The vectors; p, r and a consist of the precision, recall and accuracy respectively for each experimental trial. The higher the resultant summation of the performance measures the higher the ranking of the given performance. Therefore the trails that had the highest performance weighted sum were considered to have performed the best. The $rank_{x,i}$ is calculated per repository trial and the best performance ($best_x$) per repository is calculated.

$$\begin{aligned} rank_{x,i} &= w_{pi} \times p_i + w_{ri} \times r_i + w_{ai} \times a_i \\ best_x &= max(rank_x) \end{aligned} \tag{5.9}$$

While the weighted sum did optimize each value, a common issue was that often a repository would have one or two parameter perform well while the remaining performed poorly. For example blockly-android had the maximum value for recall but 0.51 for precision and accuracy. The weighted sum of this vector would be $0.51 \times 0.25 + 1.0 \times 0.5 + 0.51 \times 0.25 = 0.755$. However in the event that another

Repository	AI	Feature Set	SWR	Setup	Precision	Recall	Accuracy
acra	SVM	2	80	exp 1	0.74	0.92	0.8
arquillian-core	RF	3	90	exp 2	0.53	0.98	0.55
blockly-android	SVM	2	60	exp 1	0.51	1.0	0.51
brave	RF	2	110	exp 1	0.59	0.97	0.65
cardslib	SVM	2	120	exp 1	0.5	1.0	0.5
dagger	RF	3	90	exp 2	0.5	1.0	0.5
deeplearning4j	RF	2	70	exp 1	0.55	0.96	0.58
fresco	RF	2	60	exp 1	0.52	1.0	0.53
governator	RF	2	60	exp 1	0.57	0.81	0.6
greenDAO	SVM	4	90	exp 2	0.5	1.0	0.5
http-request	RF	2	80	exp 1	0.87	0.79	0.84
ion	RF	2	90	exp 2	0.72	0.73	0.72
jadx	SVM	2	130	exp 1	0.55	0.82	0.58
mapstruct	SVM	2	70	exp 1	0.6	0.88	0.65
nettosphere	RF	2	110	exp 1	0.63	0.67	0.63
parceler	SVM	1	90	exp 2	0.57	0.92	0.61
retrolambda	SVM	2	130	exp 1	0.5	1.0	0.5
ShowcaseView	SVM	1	90	exp 2	0.73	0.89	0.78
smile	SVM	2	70	exp 1	0.5	1.0	0.5
spark	SVM	4	90	exp 2	0.5	1.0	0.5
storm	RF	2	60	exp 1	0.61	0.89	0.66
tempto	RF	2	120	exp 1	0.53	0.73	0.55
yardstick	SVM	2	70	exp 1	0.55	0.79	0.57

Table 5.23: Repository Best Performance

performance scored 0.7 for each measure, the weighted sum would be $0.7 \times 0.25 + 0.7 \times 0.5 + 0.7 \times 0.25 = 0.7$. Since the goal is for multi-variable optimization a model that is generally good on each measure is better than a model performs well with one or two but terribly for the rest. Regardless though, some generally performed poorly and that is reflected in the table with a best performance that is quite low or very bias.

The number of repositories that perform the best with SVM and RF are 11 and 12 respectively. As noted above some of the performance results are still quite low

for some of the repositories because of a lack of success for that repository for all of the trials run. All repositories did perform better than 50% which is quite a poor result since it is about as good as a coin toss. Out of the best performances, only 6 of the 23 repositories have a trial that performs better than 60% for all three measures. Of those 6 only 4 of those perform better than 70% for each performance measure. Of the top 4, http-request performed the best followed by acra, ShowcaseView and ion in descending order. Both acra and ShowcaseView performed best with use of SVM while http-request and ion perform best with RF. For both http-request and ShowcaseView the overall performance is better for RF over SVM. Likewise, ion performed better for SVM over RF. The results were more consistent between RF and SVM. The second feature set used for acra, http-request and ion to gain the best performance while for ShowcaseView the first feature set performed the best. On a final note, each of these 4 repositories are classified as long in length, small in size, small to medium in number of developers and low to medium in commit rate.

The parameters setup for each experiment proved to vary in results. In order to determine which set of parameters are the most ideal the trails for each experimental parameters are ranked. In Equation 5.10 the ranked score is calculated similar to that in Equation 5.9. However for Equation 5.10, the ranks are ordered in based on the parameter set (s) used for the trial. Therefore the weighted summation, $rank_{s,i}$ is taken for each repository with the same parameter set up s. The average, $avg_{rank_{s,i}}$, is taken for each parameter set. The best ranked parameter is set is then selected by taking the maximum avg_{rank_s}.

$$\begin{aligned} rank_{s,i} &= w_{pi} \times p_i + w_{ri} \times r_i + w_{ai} \times a_i \\ avg_{rank_s} &= \frac{\sum_i^n rank_{s,i}}{|rank_s|} \\ best_{rank_s} &= max(avg_{rank_s}) \end{aligned} \tag{5.10}$$

AI	Feature Set	SWR	Setup	Rank Average
rf	2	100	exp-1	0.65065
rf	2	110	exp-1	0.63835
rf	2	60	exp-1	0.6324
rf	2	70	exp-1	0.6291
rf	2	120	exp-1	0.62895
rf	3	90	exp-2	0.62255
rf	2	130	exp-1	0.62115
rf	2	90	exp-2	0.6127
rf	2	90	exp-1	0.61165
rf	5	90	exp-2	0.60555
rf	2	80	exp-1	0.6041
rf	1	90	exp-2	0.6033
svm	5	90	exp-2	0.596
rf	4	90	exp-2	0.5922
svm	4	90	exp-2	0.58345
svm	2	70	exp-1	0.5788
svm	2	120	exp-1	0.57755
svm	2	130	exp-1	0.57645
svm	1	90	exp-2	0.57385
svm	2	60	exp-1	0.5481
svm	3	90	exp-2	0.5428
svm	2	90	exp-1	0.538
svm	2	100	exp-1	0.5344
svm	2	90	exp-2	0.53265
svm	2	80	exp-1	0.51825
svm	2	110	exp-1	0.50445

Table 5.24: Best Parameter Performance

The complete order list of the ranked parameter sets are shown in Table 5.24. Out of all the parameter setups, RF ranked higher for all setups except for one which ranked lower than the best parameter setup for SVM. This ranked list can be considered as a guide to identify the best performing parameter setup. Also the best parameter set can be used for any new repositories collected to attempt to get the best results.

5.4 Threats to Validity

Sampling a larger set of repositories was more beneficial for the analysis of the method since performance was not consistent across all repositories. As done with the groups of repositories, more repositories allows for trends to be examined between repositories. A concerted effort was made to contrast positive results for one repository with negative results for other repositories. Such a contrast may mitigate the impact of the positive results, however provides the full context and help direct future work in this area.

Each experiment was designed to attempt to provide a robust setup to measure accurately the performance of the approach given the changes to the current factor. The setup was designed to attempt to preventing the influence of other variables beyond the independent variable. The factors that may have had an influence on the experimental results are the third experiment only sampling the extremes (best and worst).

A major concern with the final experiment was that of the sampling of the best and worst results from the previous experiments to test the use of oversampling. While the results of the use of oversampling should not be discounted, only the sampling the extremes of the previous experiment may have limited the measurable impact of the use of oversampling. This experiment could be extended to test the middle performance or even test each trail from the previous experiments.

The differences between the repositories prevented a more direct comparison between the repositories. Furthermore, as shown through the experiments, some repositories (e.g. ShowcaseView) generally performed better than other repositories (e.g. governator). This leads to the conclusion that certain repository related factors have a large impact on the performance of the approach. Further investigation into these

repository specific factors could lead to improved results for the approach.

The repositories sampled may also not be representative of OSS repositories more generally as we first restricted the repositories in two respects:

1. Repositories contained a majority Java source code.

2. Repositories contained c commits where $300 < c < 4000$.

Therefore while the experimentation was applied to several projects a large number of repositories do not meet these requirements. Furthermore, only OSS where considered and experimented on and therefore these experiments may be unrepresentative of closed source repositories as well. The major mitigating factor in both of these cases is that each repository prediction model is self contained and not reliant on any other repository for training.

Chapter 6

Conclusions

6.1 Summary

We proposed a method that leverages the commit history to predict future changes within the repository. The data used for predictions was collected from Open Source Software (OSS) repositories on GitHub. The data was then visualized through several techniques to help identify key features for use in the prediction model. The features were selected and a model was created to predict whether change will occur in the short term of 5 commits. Three experiments were conducted on the approach using 23 different OSS repositories developed in Java. The experiments investigated the three different factors; sampling range, model features and data set balancing to identify measure the impact on the performance of the approach. The results of the experiments show that while the Sample Window Range (SWR) had a strong impact on the performance, the repositories themselves often had internal factors which caused differences in performance.

6.2 Contributions

The contributions of this work are:

1. Determined which factors more strongly influence the performance of the predictions approach.

2. Out of the three factors investigated, the SWR proved to have the greatest impact for both Support Vector Machine (SVM) and Random Forest (RF).

3. Providing an approach that with some success can predict future changes within a repository using the commit data. Both SVM and RF are viable for a few repositories such as acra, http-request and ShowcaseView.

6.3 Limitations

The limitations placed on this approach often are divided into two categories; limitations used to simplify the approach and limitations inherent to the approach. The limitations used to simplify the approach are:

- Repositories are required to have a commit size of $300 < c < 4000$. The lower bound helped prevent experimentation on very small repositories which would not have had enough data. Likewise the upper bound helped restrict experimentation on huge repositories which will take longer to train and provide a more difficult challenge when modeling change.

- Repositories must consist of at least 75% Java source code. The prediction model feature collection process required identifying language specific details and would require a reimplementation for any new language.

- Repositories had to be developed on GitHub and openly available. Since the goal was to extract the data from an entire repository the availability was essential. GitHub provides an easy to use interface for collecting repository data and used to collect the data. Another method for collection or another source could be devised as long as the method collects the necessary data and the source provides the necessary data.

While some of these restrictions are easily circumvented some are also inherent to the implementation. For example the second two would require fairly substantial reimplementation to overcome these limitations. These are however possible since creating a method for collecting from different sources would open the approach to more repositories and allow for further use. Likewise, implementations done in other languages would allow for more repositories to make use this approach.

The other set of limitations outlines ones that are internal to the approach and are only discovered through experimentation. These limitations are:

- Repository internal factors play a large role in how well the repository performs. The experimental results showed that while some repositories may perform well others classified similarly will perform very differently. These factors were not effectively controlled by the parameters and therefore the results are unpredictable.

- The prediction model is very sensitive to differences within a given repository which is easily shown in the variation between repository results for a given set of parameters. Therefore the ideal parameters for a given repository are only ideal for the current section of the data set investigated.

- Finding the optimal set of parameters for creating the prediction model is very difficult since these parameters will vary per repository. The extend of the

impact of the parameters while investigated was not fully quantified. Therefore in order to find a better solution for a given repository tunning of the parameters is necessary. Finally, while a good solution may be found better solutions may be available but difficult to find.

The issues outlined here are more difficult to overcome and may require a redesign of the approach to address. Specifically, the first two limitations are relate to impact of repository specific characteristics on the performance. Mitigating these characteristics with parameters would help allow for easier configuration of the method that achieves better results. Finally, achieving an optimal solution is very difficult because of how long it takes to train the model and a general rule is difficult to find with such differences between repositories.

6.4 Future Work

This work provided a starting point for predicting future code changes within a source code repository based on change history. In the future this work could be extended in the following directions:

- **Language** – Our approach focused primarily on Java OSS repositories found on GitHub. The approach could be extended to predict changes in other languages and also multi-language repositories. We observed that repositories that shared similar characteristics tended to perform similarly (e.g. ShowcaseView, http-request and ion).

- **Training Feature Weighting** – The experiments used the same weighting for all features used to train the model. The use of different weighting could help fine tune the approach and improve the results.

- **Multi-Classification** – Both SVM and RF relied on making a simple binary classification and could be extended to make multi-classifications to provide more refined results. One such multi-classifications would be the size of change predicted to occur in the five commits.

- **Predictive Model** – Use of different machine learning or Artificial Intelligence (AI) techniques to perform the predictions could influence the performance of a given project. Experimenting with a wider variety of techniques could achieve better results and is an avenue for further research. Specifically, use of deep learning algorithms could prove useful, given this approach looks at a single repository. However in the case of larger projects, such as storm and deeplearning4j, the machine learning techniques tended to perform poorly. Using a regressors to train on a larger data set could allow for stronger results, given that a regressor tends to require a larger data sample to train on to be more effective.

- **Data Sources** – The approach could be extended to integrate source code metrics to help improve the model.

A more expansive experimental result could be conducted by sampling from a larger number of repositories. This could help provide more insights into the prediction model and best practices with respect to training a new repository. Finally a more extensive look at the other factors that were involved in the approach would be useful in further improving the performance of the approach and predicting changes within the development of a application.

Bibliography

[1] Alam, M. S., and Vuong, S. T. Random Forest Classification for Detecting Android Malware. In *Proceedings of the Green Computing and Communications, IEEE Internet of Things, IEEE Cyber, Physical and Social Computing* (2013), pp. 663–669.

[2] Antón, J. C. Á., Nieto, P. J. G., Viejo, C. B., and Vilán, J. A. V. Support Vector Machines Used to Estimate the Battery State of Charge. *IEEE Transactions on Power Electronics 28*, 12 (2013), 5919 – 5926.

[3] Bantelay, F., Zanjani, M. B., and Kagdi, H. Comparing and combining evolutionary couplings from interactions and commits. In *Proceedings of the Working Conference on Reverse Engineering, WCRE* (2013), pp. 311–320.

[4] Bhattacharyya, S., Jha, S., Tharakunnel, K., and Westland, J. C. Data mining for credit card fraud: A comparative study. *Decision Support Systems 50*, 3 (2011), 601–613.

[5] Bieman, J., Andrews, A., and Yang, H. Understanding change-proneness in OO software through visualization. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension, 2003.* (2003), pp. 44 – 53.

[6] Burbidge, R., Trotter, M., Buxton, B., and Holden, S. Drug design by machine learning : support vector machines for pharmaceutical data analysis. *Computers and Chemistry 26*, 1 (2001), 5 – 14.

[7] Canfora, G., Cerulo, L., and Di Penta, M. Identifying changed source code lines from version repositories. In *Proceedings of the 4th International Workshop on Mining Software Repositories, MSR 2007* (2007), pp. 14 – 22.

[8] Chaturvedi, K. K., Kapur, P. K., Anand, S., and Singh, V. B. Predicting the complexity of code changes using entropy based measures. *International Journal of System Assurance Engineering and Management 5*, 2 (2014), 155–164.

[9] Collberg, C., Kobourov, S., Nagra, J., Pitts, J., and Wampler, K. A system for graph-based visualization of the evolution of software. In *Proceedings of the 2003 ACM symposium on Software visualization - SoftVis '03* (2003), pp. 77 – 86.

[10] De Souza, C. R., Quirk, S., Trainer, E., and Redmiles, D. F. Supporting collaborative software development through the visualization of socio-technical dependencies. *2007 International ACM Conference on Supporting Group Work, GROUP'07, November 4, 2007 - November 7, 2007* (2007), 147–156.

[11] Dit, B., Holtzhauer, A., Poshyvanyk, D., and Kagdi, H. A dataset from change history to support evaluation of software maintenance tasks. In *Proceedings of the 10th Working Conference on Mining Software Repositories* (2013), pp. 131–134.

[12] Erturk, E., and Sezer, E. A. A comparison of some soft computing methods for software fault prediction. *Expert Systems with Applications 42*, 4 (2015), 1872–1879.

[13] Gall, H. C., and Lanza, M. Software Evolution : Analysis and Visualization. In *Proceedings of the 28th international conference on Software engineering* (2006), pp. 1055–1056.

[14] Giger, E., Pinzger, M., and Gall, H. C. Can we predict types of code changes? An empirical analysis. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR), 2012* (2012), pp. 217–226.

[15] Gilbert, E., and Karahalios, K. CodeSaw: A social visualization of distributed software development. In *Proceedings of the 11th IFIP TC 13 international conference on Humancomputer interactionVolume Part II* (2007), pp. 303–316.

[16] Gondra, I. Applying machine learning to software fault-proneness prediction. *Journal of Systems and Software 81*, 2 (2008), 186–195.

[17] Gonzalez, A., Theron, R., Telea, A., and Garcia, F. J. Combined Visualization of Structural and Metric Information for Software Evolution Analysis. In *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops* (2009), pp. 25–29.

[18] Granitto, P. M., Gasperi, F., Biasioli, F., and Furlanello, C. Modern data mining tools in descriptive sensory analysis: A case study with a Random forest approach. *Food Quality and Preference 18*, 4 (2007), 681–689.

[19] Guo, L., Ma, Y., Cukic, B., and Singh, H. Robust Prediction of Fault-Proneness by Random Forests. In *Proceedings of the 15th International Symposium on Software Reliability Engineering, 2004* (2004), pp. 417–428.

[20] Hassan, A. E. Mining software repositories to assist developers and support managers. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance* (2006), pp. 339–342.

[21] Hassan, A. E., and Holt, R. C. Predicting change propagation in software systems. In *20th IEEE International Conference on Software Maintenance, 2004* (2004), pp. 284–293.

[22] Hemmati, H., Nadi, S., Baysal, O., Kononenko, O., Wang, W., Holmes, R., and Godfrey, M. W. The MSR cookbook: Mining a decade of research. In *Proceedings of the 10th Working Conference on Mining Software Repositories* (2013), pp. 343–352.

[23] Huang, C.-l., Chen, M.-c., and Wang, C.-j. Credit scoring with a data mining approach based on support vector machines. *Expert Systems with Applications 33*, 4 (2007), 847–856.

[24] Jalbert, K., and Bradbury, J. S. Predicting mutation score using source code and test suite metrics. In *Proceedings of the 1st International Workshop on Realizing AI Synergies in Software Engineering* (jun 2012), Ieee, pp. 42–46.

[25] Kagdi, H., and Maletic, J. I. Combining single-version and evolutionary dependencies for software-change prediction. In *Proceedings of the 4th International Workshop on Mining Software Repositories, MSR 2007* (2007).

[26] Keim, D. a. Information visualization and visual data mining. *IEEE Transactions on Visualization and Computer Graphics 8*, 1 (2002), 1–8.

[27] Khoshgoftaar, T. M., Golawala, M., and Van Hulse, J. An Empirical Study of Learning from Imbalanced Data Using Random Forest. In *Proceedings of the 19th IEEE International Conference on Tools with Artificial Intelligence* (2007), pp. 310–317.

[28] Kim, K.-j. Financial time series forecasting using support vector machines. *Neurocomputing 55*, 1-2 (2003), 307–319.

[29] Kim, S., Whitehead, E. J., and Zhang, Y. Classifying Software Changes : Clean or Buggy ? *IEEE Transactions on Software Engineering 34*, 2 (2008), 181–197.

[30] Lanza, M., Ducasse, S., Gall, H., and Pinzger, M. CodeCrawler - an information visualization tool for program comprehension. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* (2005), no. July 2015, pp. 672 – 673.

[31] Maletic, J. I., and Collard, M. L. Supporting source code difference analysis. In *Proceedings of the 20th IEEE International Conference on Software Maintenance* (2004), pp. 210–219.

[32] Malhotra, R. A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing 27*, C (2015), 504–518.

[33] Moeyersoms, J., Fortuny, E. J. D., Dejaeger, K., Baesens, B., and Martens, D. Comprehensible software fault and effort prediction : A data mining approach. *Journal of Systems & Software 100* (2015), 80–90.

[34] Moser, R., Pedrycz, W., and Succi, G. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In

Proceedings of the 30th international conference on Software engineering (2008), pp. 181–190.

[35] Murphy, C., Kaiser, G., and Arias, M. An Approach to Software Testing of Machine Learning Applications. In *Proceedings of the 19th International Conference on Software Engineering & Knowledge Engineering* (2007), pp. 167 – 172.

[36] Nagappan, N., and Ball, T. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement, 2007.* (2007), pp. 364–373.

[37] Neuhaus, S., Zimmermann, T., Holler, C., and Zeller, A. Predicting Vulnerable Software Components. In *Proceedings of the 14th ACM conference on Computer and communications security* (2007), pp. 529–540.

[38] Ogawa, M., and Ma, K.-L. StarGate: A Unified, Interactive Visualization of Software Projects. *2008 IEEE Pacific Visualization Symposium 3*, 11 (2008), 191 – 198.

[39] Ogawa, M., and Ma, K.-L. Software Evolution Storylines. In *Proceedings of the 5th international symposium on Software visualization* (2010), pp. 35 – 42.

[40] Shivaji, S., Whitehead, E. J., Akella, R., and Kim, S. Reducing Features to Improve Code Change Based Bug Prediction. *IEEE Transactions on Software Engineering 39*, 4 (2013), 552 – 569.

[41] Sisman, B., and Kak, A. C. Incorporating version histories in Information Retrieval based bug localization. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories* (jun 2012), Ieee, pp. 50–59.

[42] Snipes, W., Robinson, B., and Murphy-Hill, E. Code hot spot: A tool for extraction and analysis of code change history. In *Proceedings of the 27th IEEE International Conference on Software Maintenance* (2011), pp. 392–401.

[43] Thwin, M. M. T., and Quah, T.-S. Application of neural networks for software quality prediction using object-oriented metrics. *Journal of Systems and Software 76*, 2 (2005), 147–156.

[44] Verikas, A., Gelzinis, A., and Bacauskiene, M. Mining data with random forests: A survey and results of new tests. *Pattern Recognition 44*, 2 (2011), 330–349.

[45] Ying, A. T. T., Murphy, G. C., Ng, R., and Chu-Carroll, M. C. Predicting Source Code Changes by Mining Change History. *IEEE Transactions on Software Engineering 30*, 9 (2004), 574–586.

[46] Yu, G., Yaun, J., and Liu, Z. Unsupervised random forest indexing for fast action search. In *Computer Vision and Pattern Recognition* (2011), pp. 865 – 872.

[47] Zeng, J., and Qiao, W. Short-term solar power prediction using a support vector machine. *Renewable Energy 52* (2016), 118–127.

[48] Zimmermann, T., Weissgerber, P., Diehl, S., and Zeller, A. Mining Version Histories to Guide Software Changes. *IEEE Transactions on Software Engineering 31*, 6 (2005), 429–445.

Appendix A

Experimental Data

A.1 Experiment 1

A.1.1 Support Vector Machine

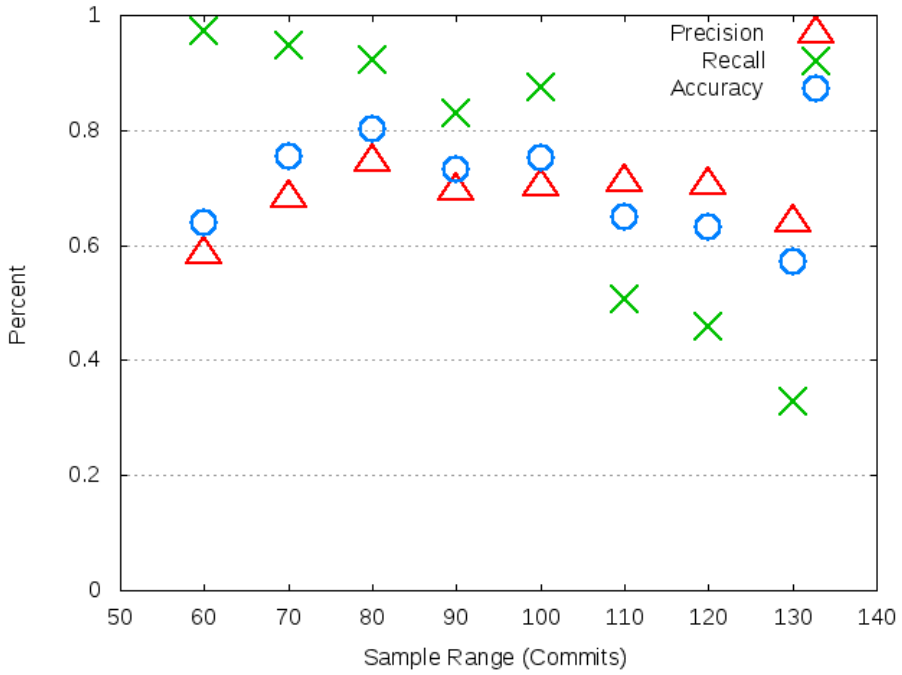


Figure A.1: Sample Window Range (SWR) for acra using Support Vector Machine (SVM)

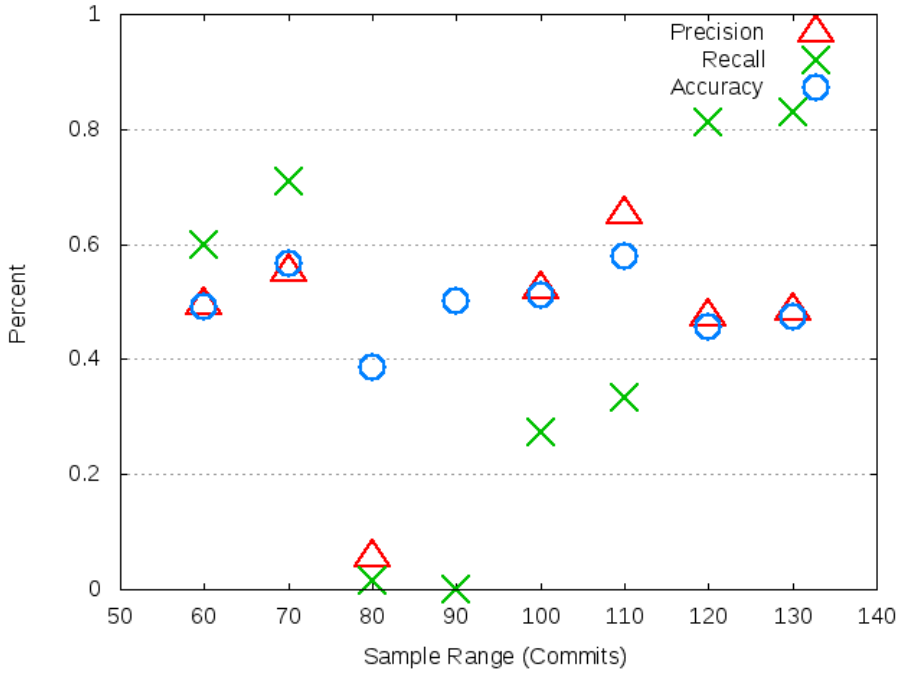


Figure A.2: SWR for arquillian-core using SVM

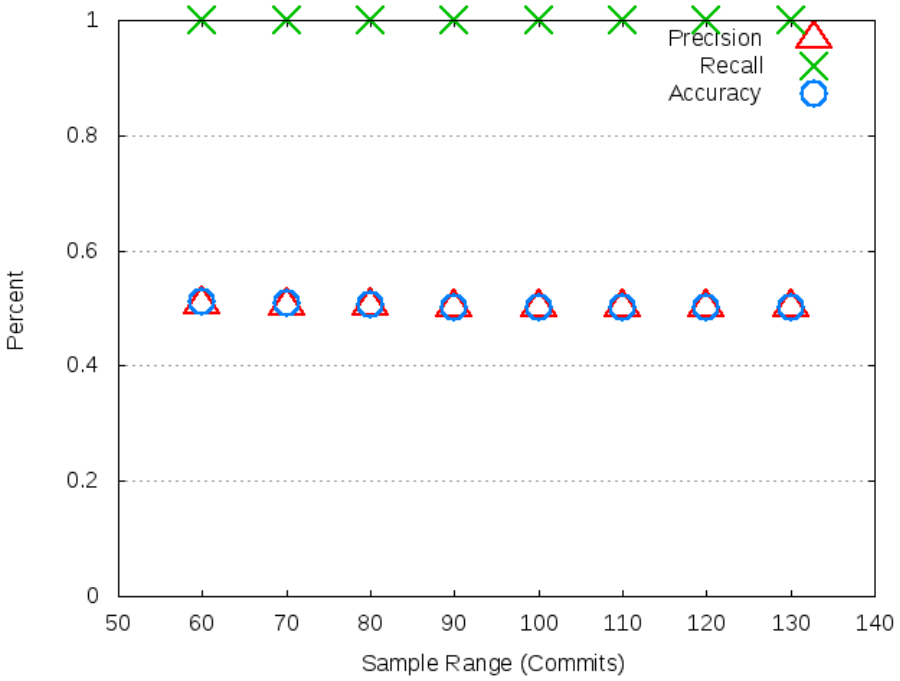


Figure A.3: SWR for blockly-android using SVM

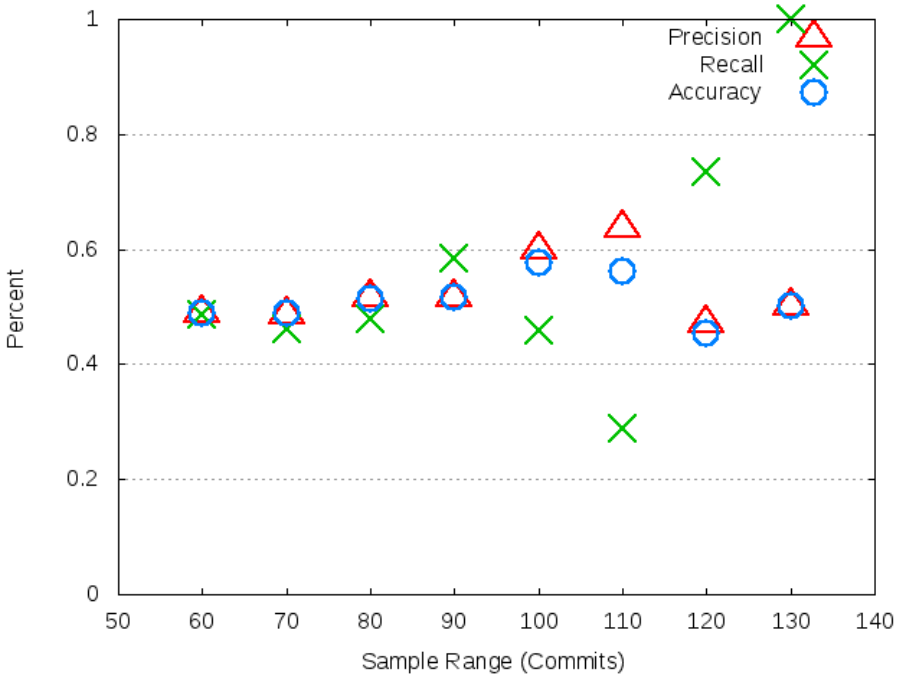


Figure A.4: SWR for brave using SVM

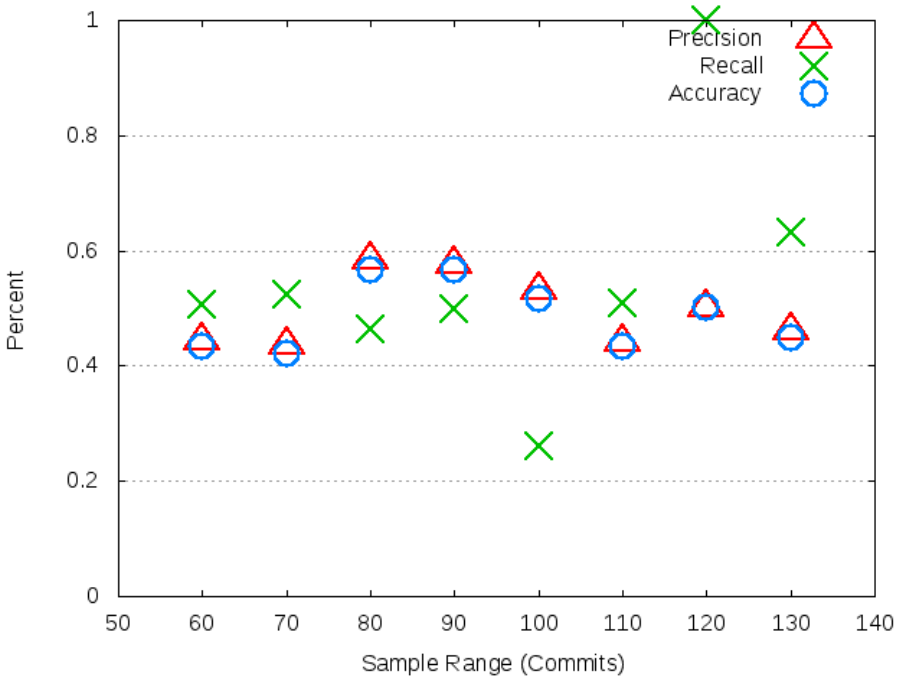


Figure A.5: SWR for cardslib using SVM

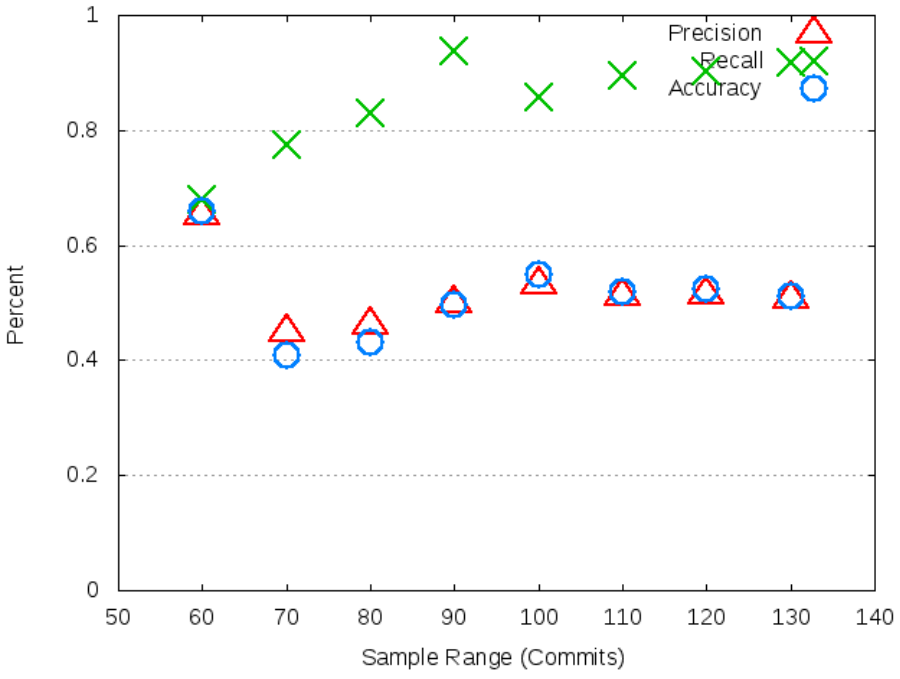


Figure A.6: SWR for dagger using SVM

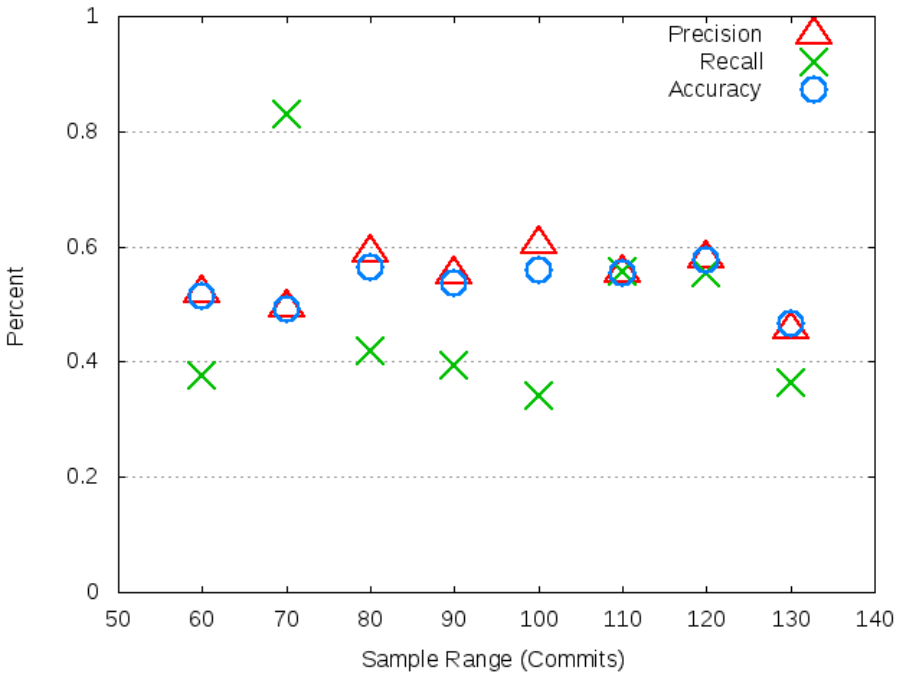


Figure A.7: SWR for deeplearning4j using SVM

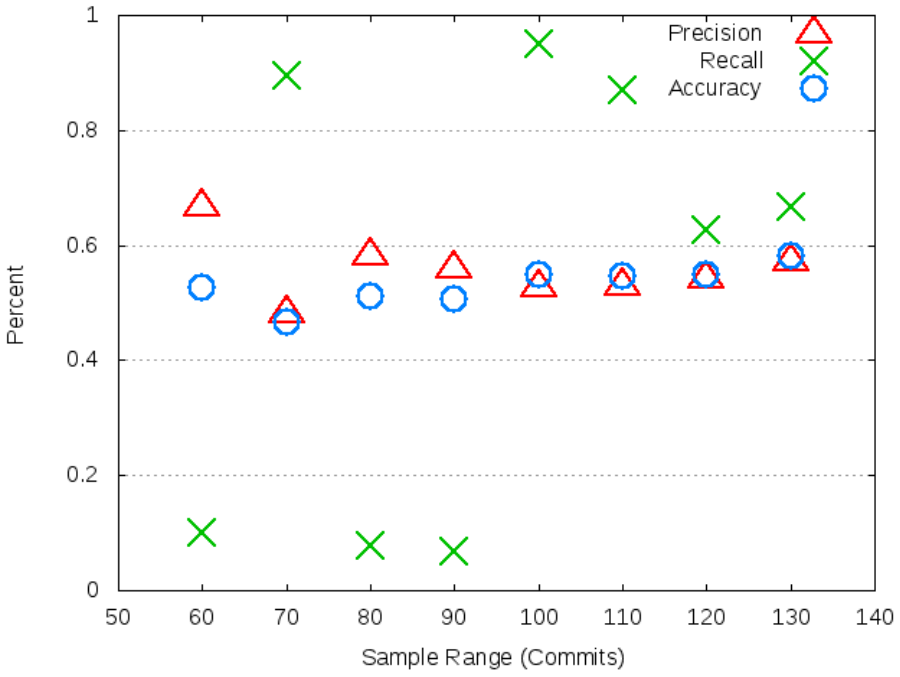


Figure A.8: SWR for fresco using SVM

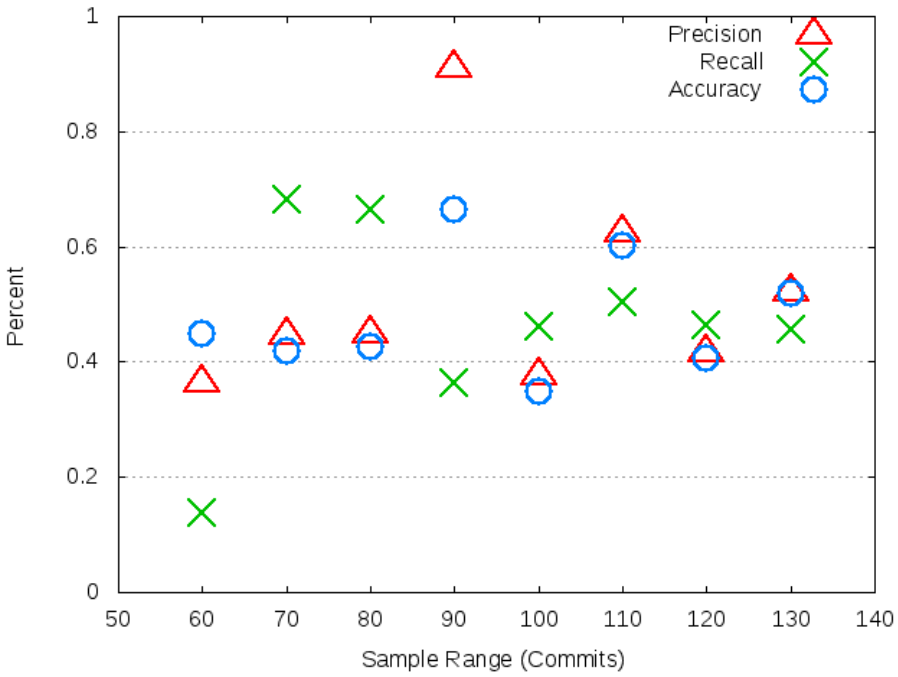


Figure A.9: SWR for governator using SVM

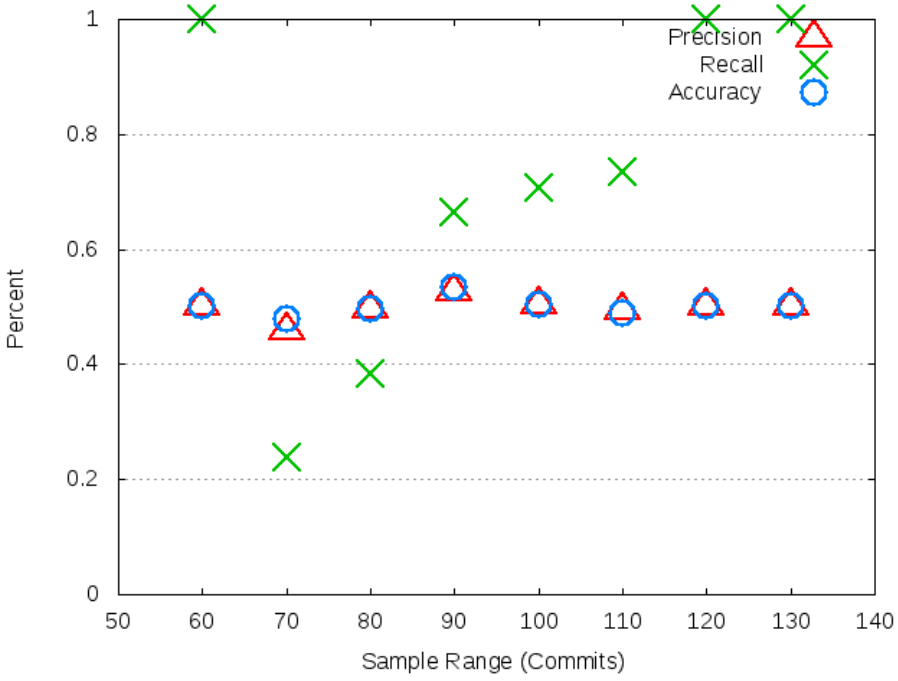


Figure A.10: SWR for greenDAO using SVM

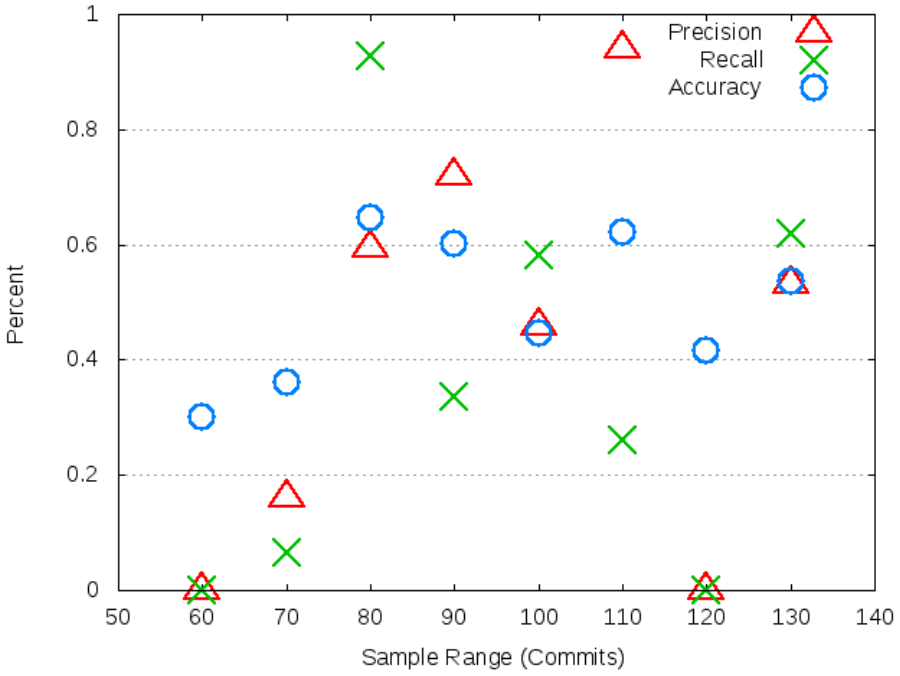


Figure A.11: SWR for http-request using SVM

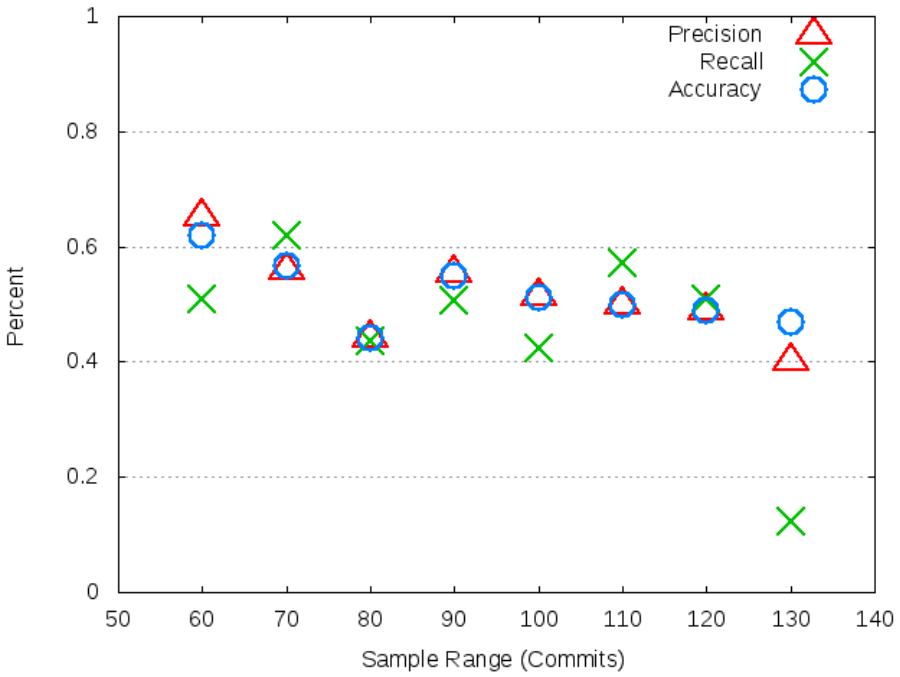


Figure A.12: SWR for ion using SVM

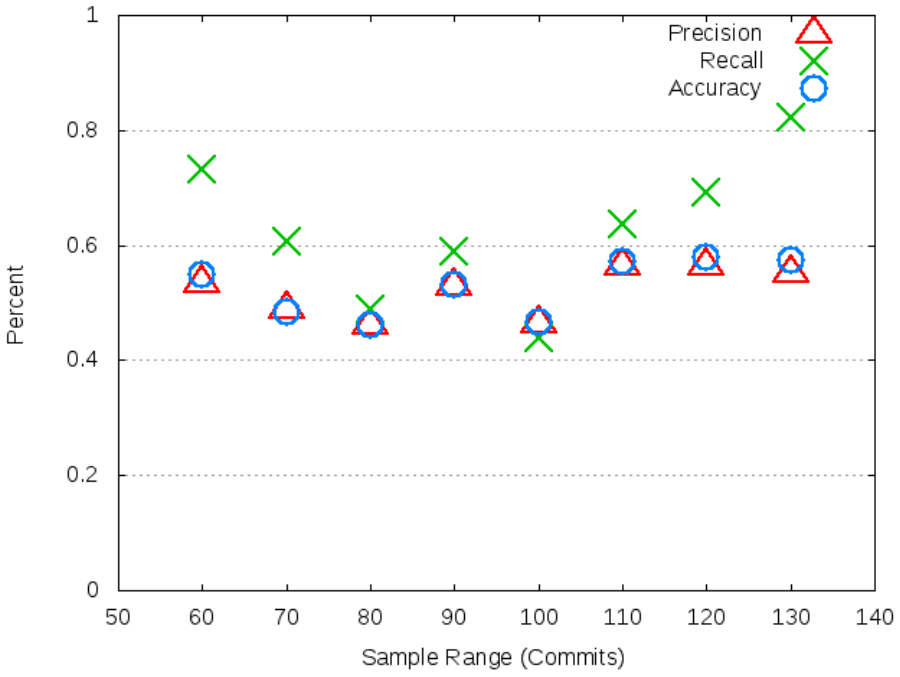


Figure A.13: SWR for jadx using SVM

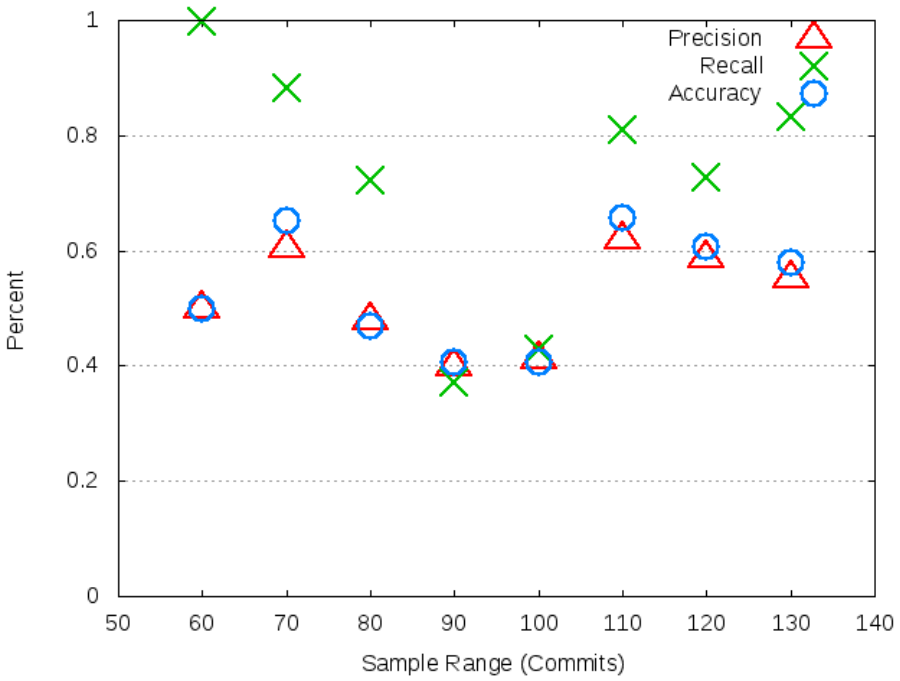


Figure A.14: SWR for mapstruct using SVM

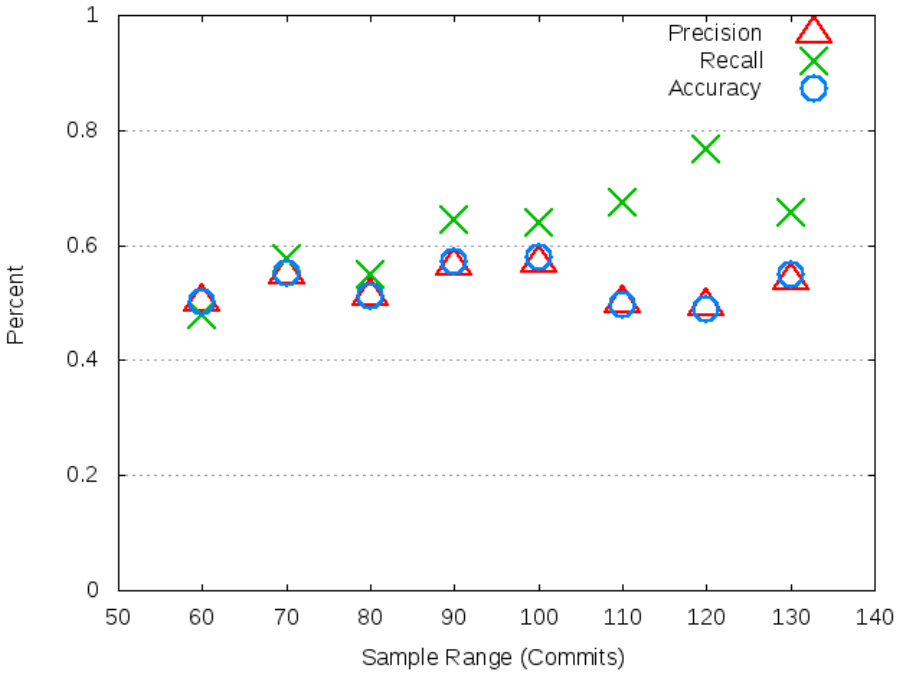


Figure A.15: SWR for nettosphere using SVM

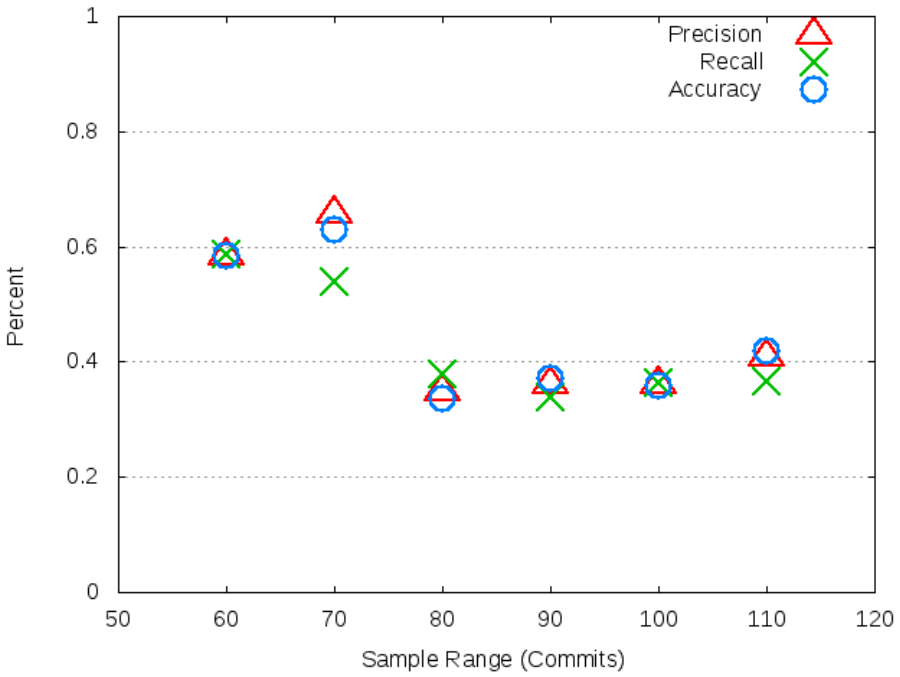


Figure A.16: SWR for parceler using SVM

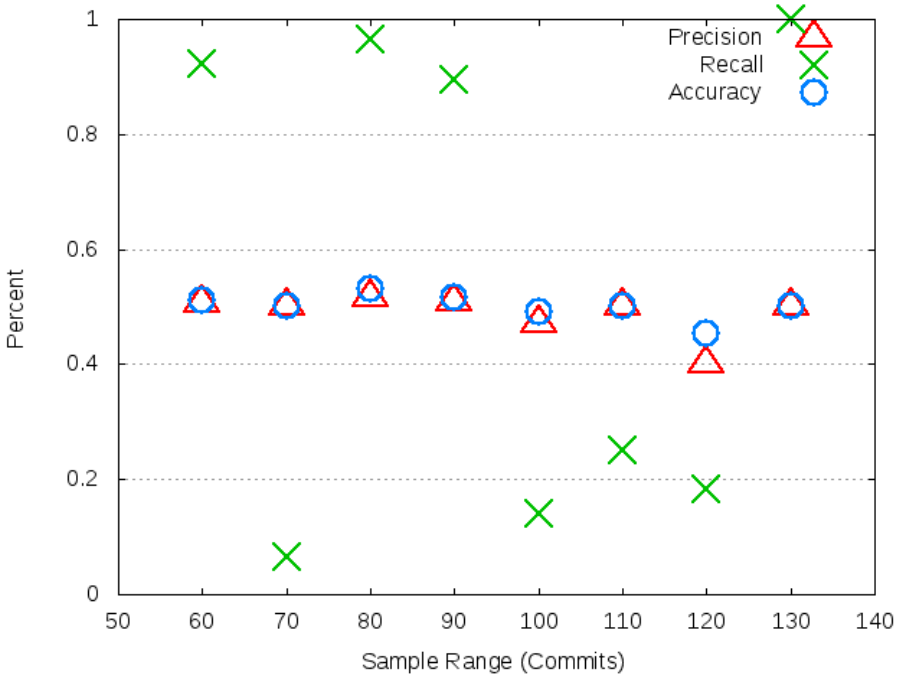


Figure A.17: SWR for retrolambda using SVM

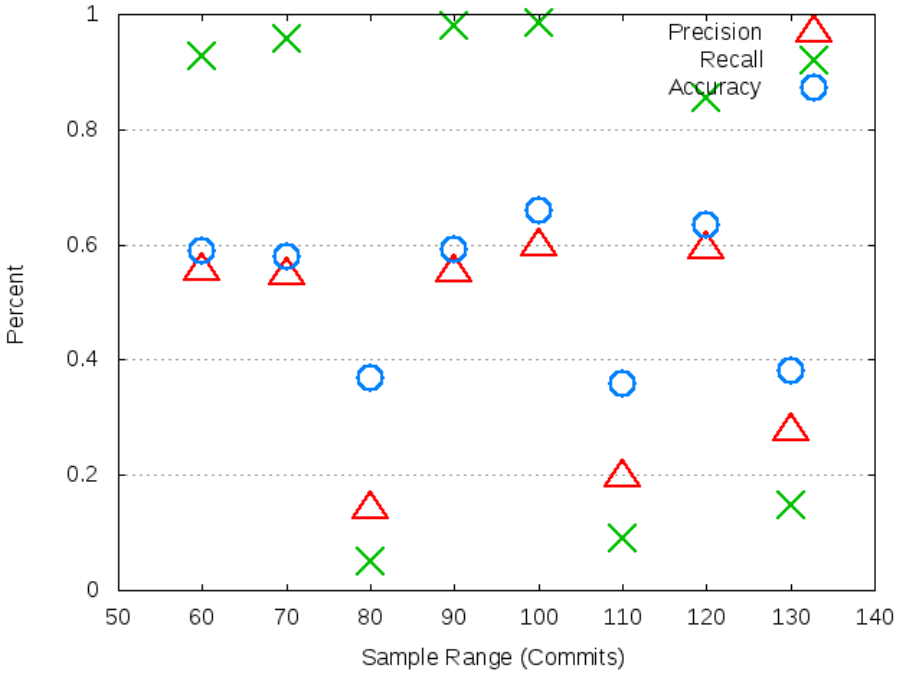


Figure A.18: SWR for ShowcaseView using SVM

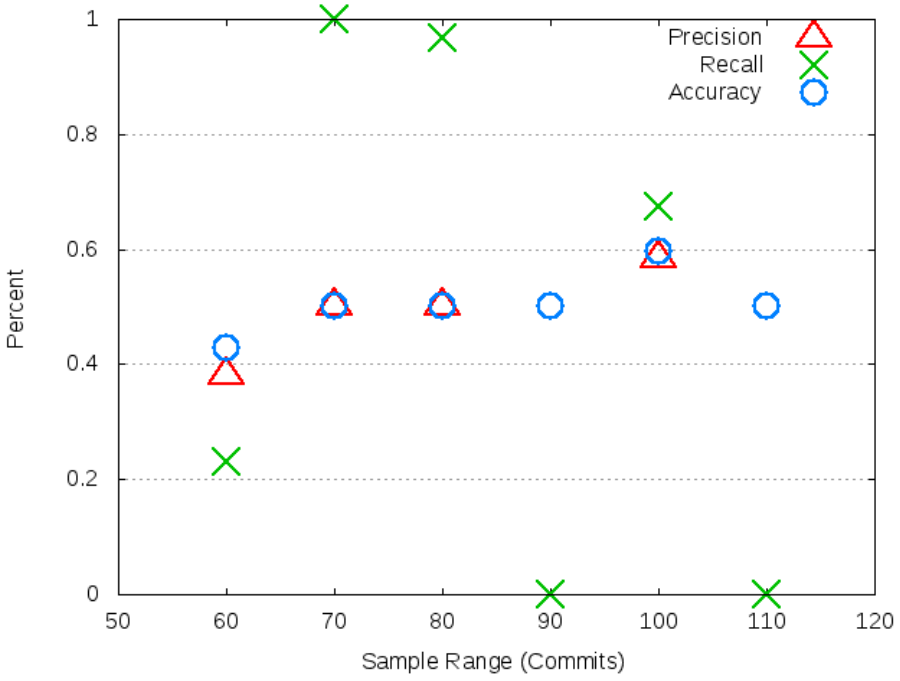


Figure A.19: SWR for smile using SVM

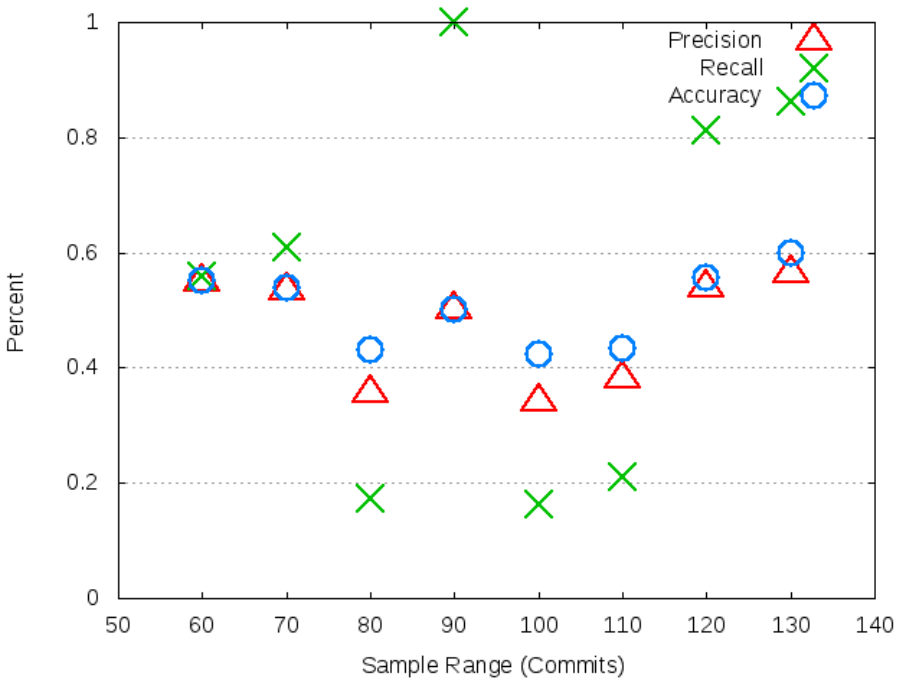


Figure A.20: SWR for spark using SVM

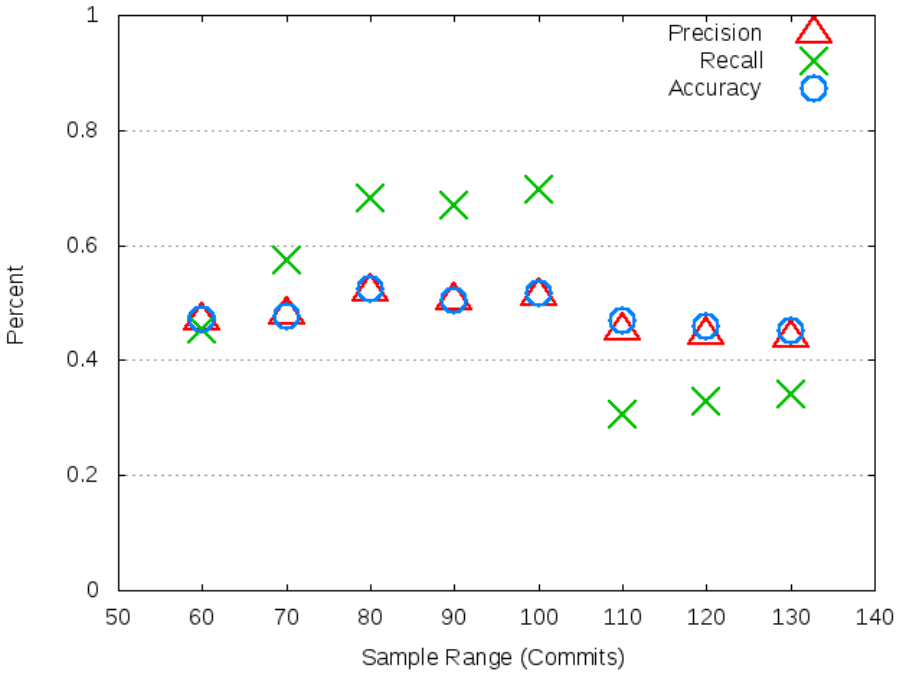


Figure A.21: SWR for storm using SVM

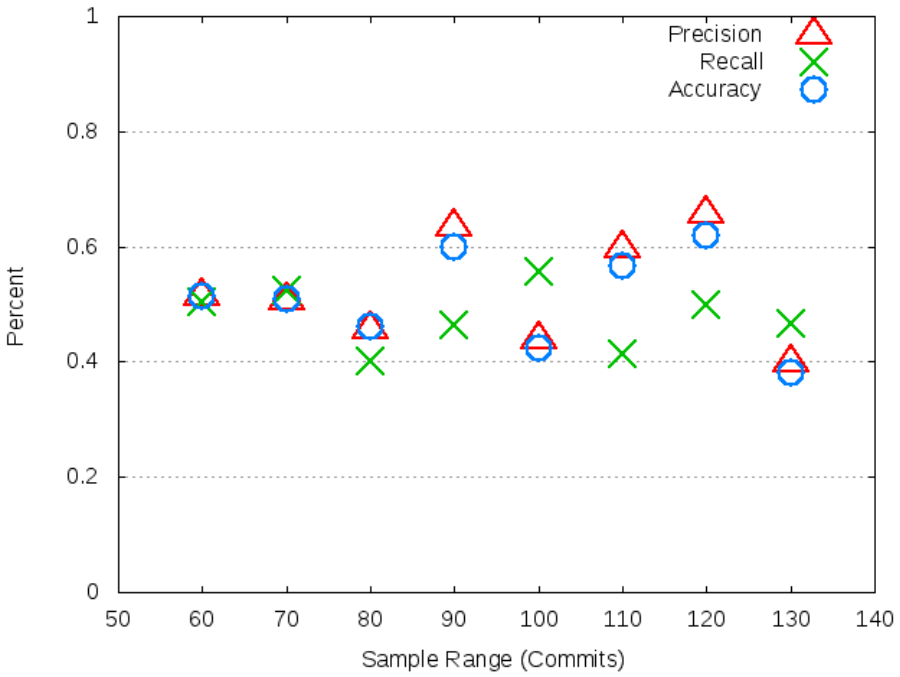


Figure A.22: SWR for tempto using SVM

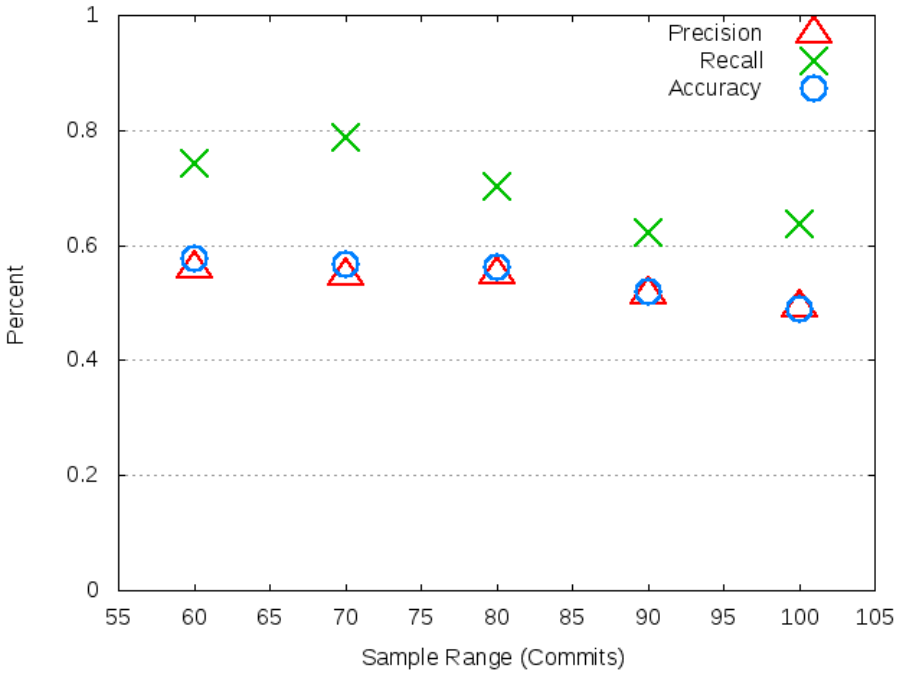


Figure A.23: SWR for yardstick using SVM

A.1.2 Random Forest

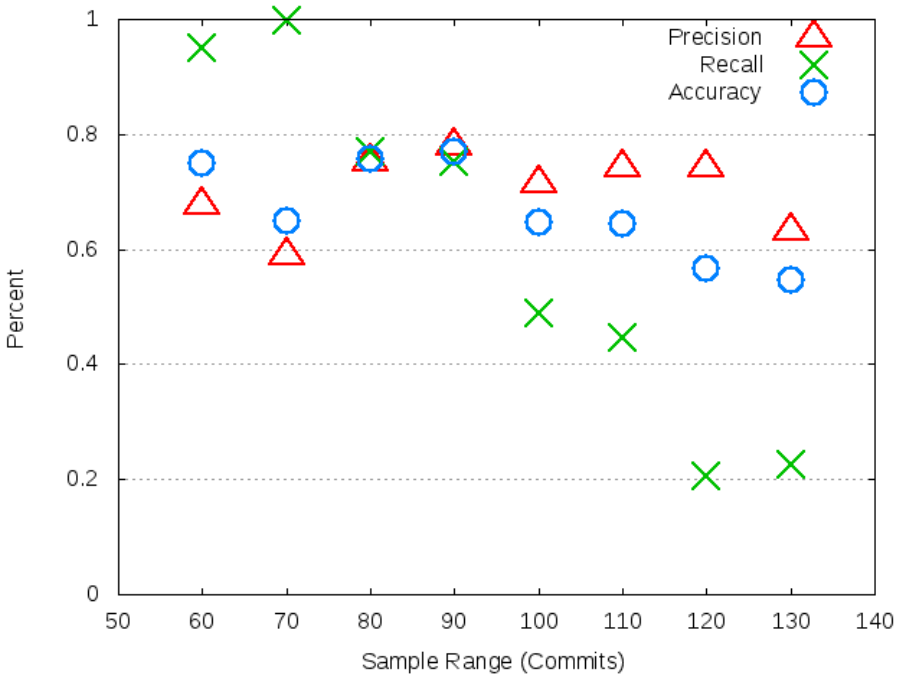


Figure A.24: SWR for acra using Random Forest (RF)

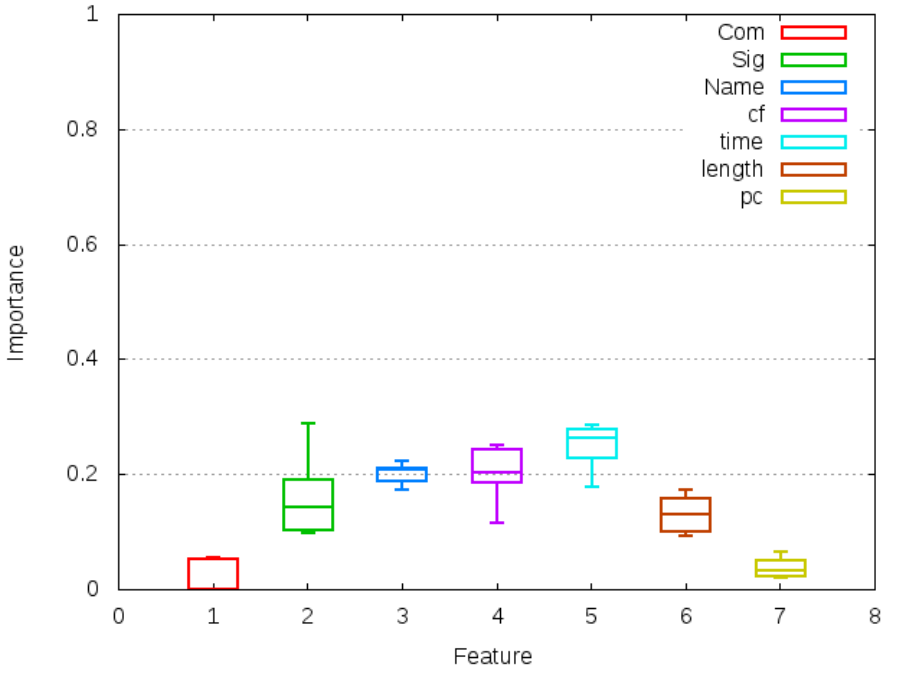


Figure A.25: Feature Importance SWR for acra using RF

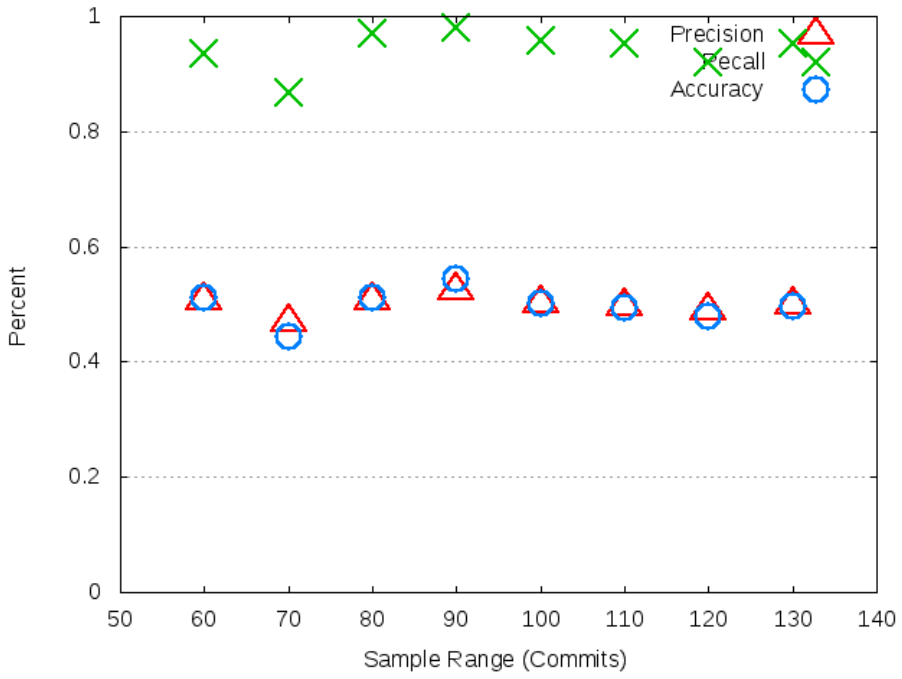


Figure A.26: SWR for arquillian-core using RF

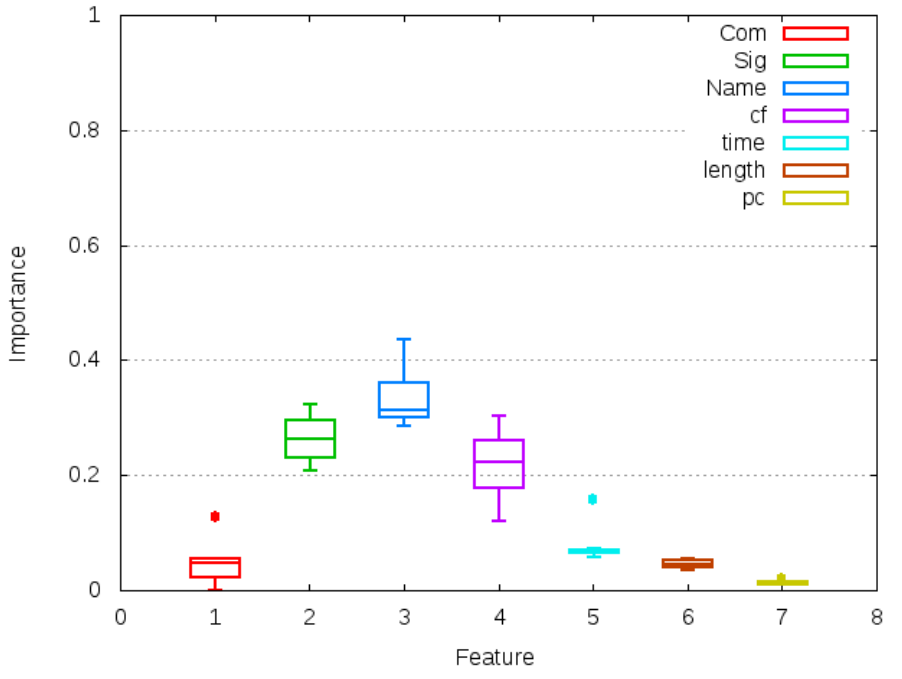


Figure A.27: Feature Importance SWR for arquillian-core using RF

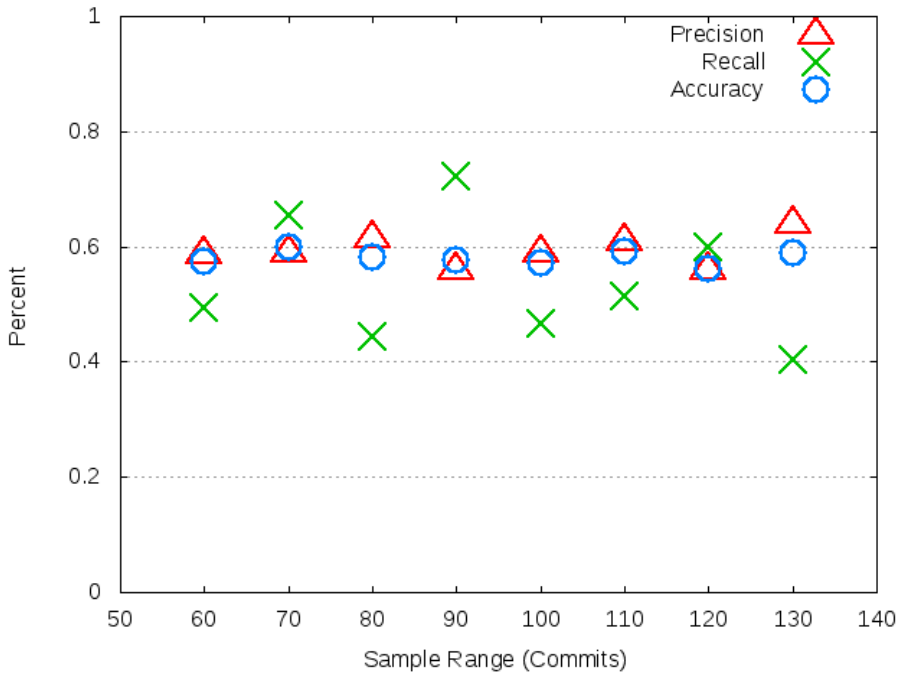


Figure A.28: SWR for blockly-android using RF

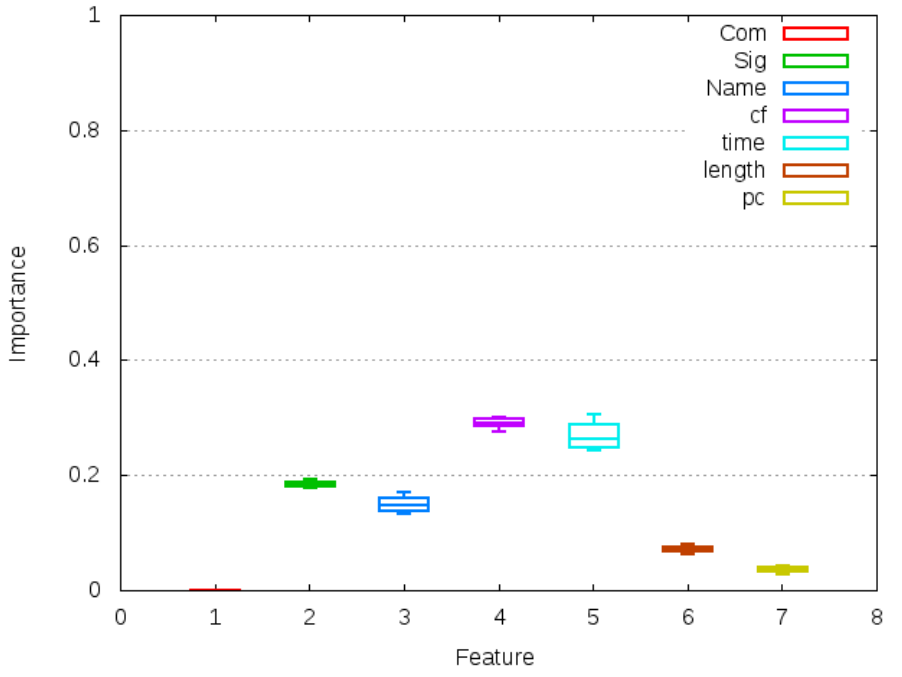


Figure A.29: Feature Importance SWR for blockly-android using RF

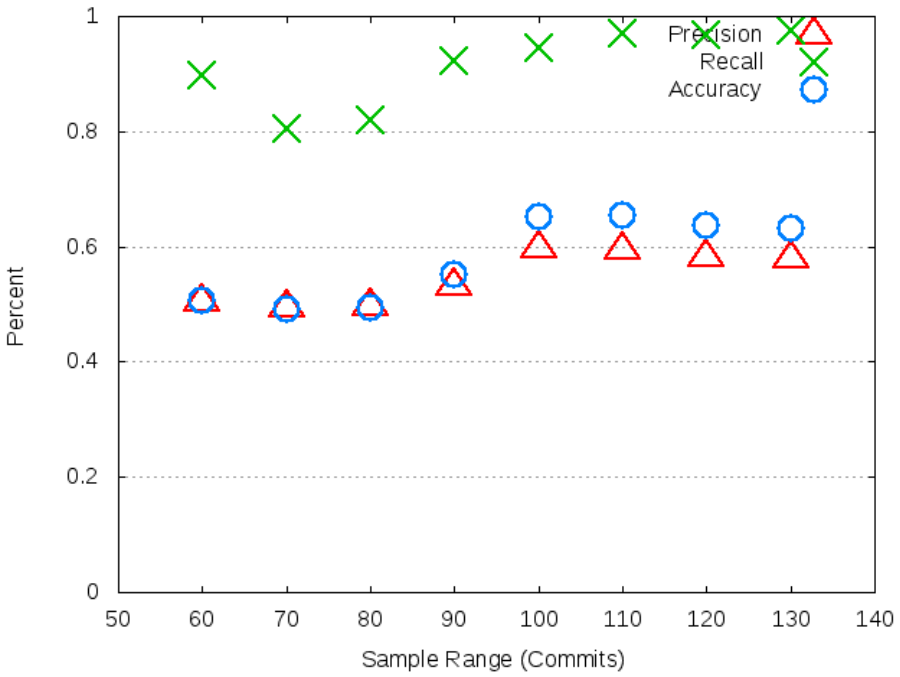


Figure A.30: SWR for brave using RF

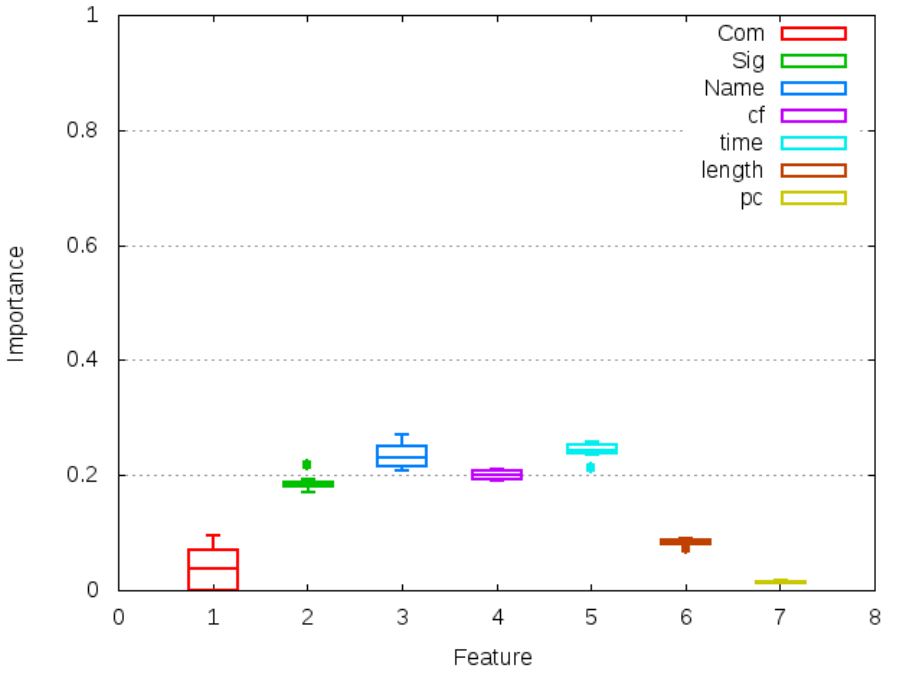


Figure A.31: Feature Importance SWR for brave using RF

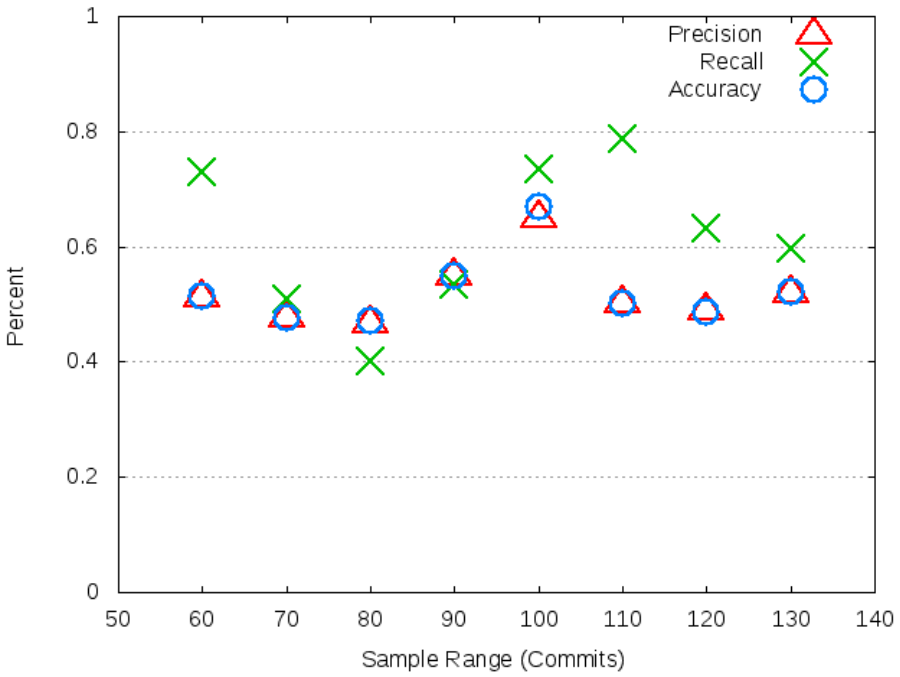


Figure A.32: SWR for cardslib using RF

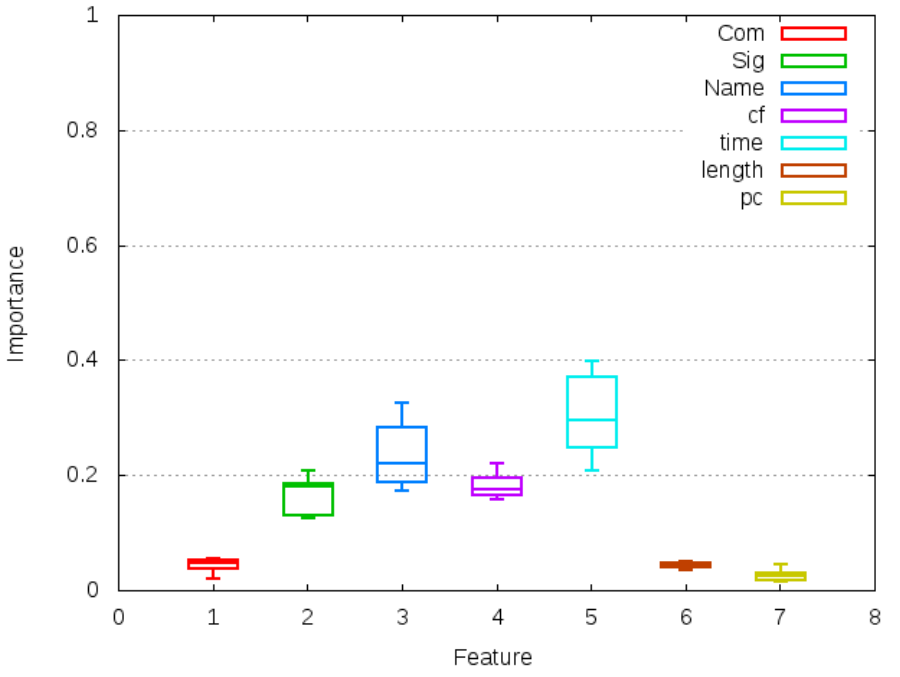


Figure A.33: Feature Importance SWR for cardslib using RF

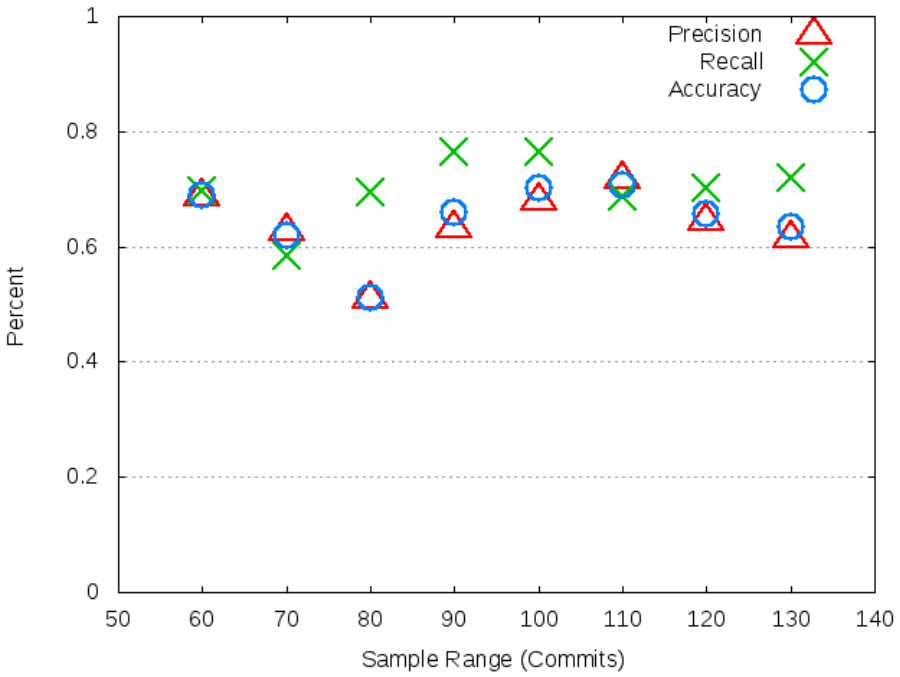


Figure A.34: SWR for dagger using RF

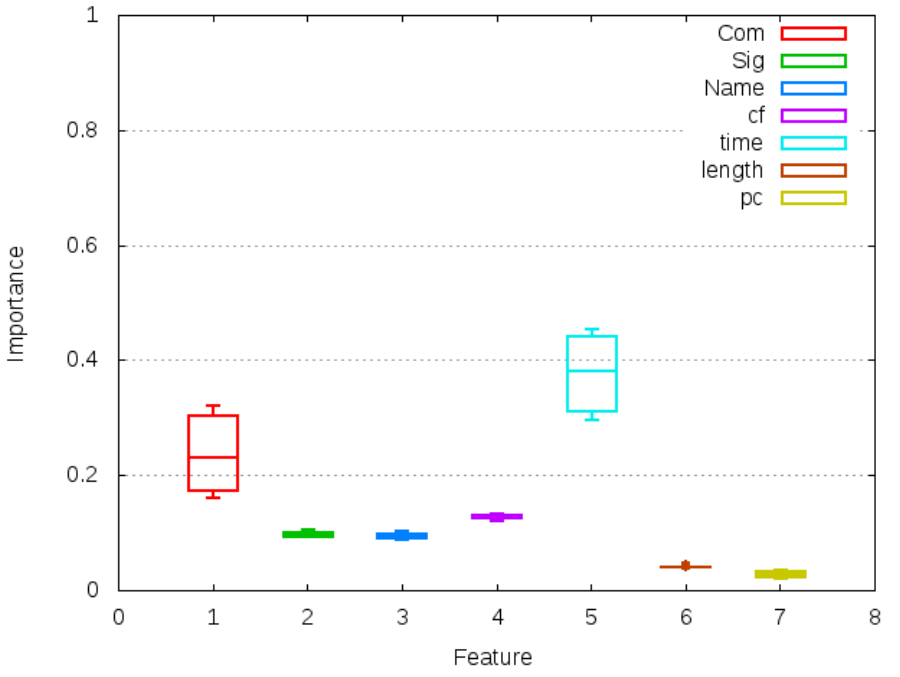


Figure A.35: Feature Importance SWR for dagger using RF

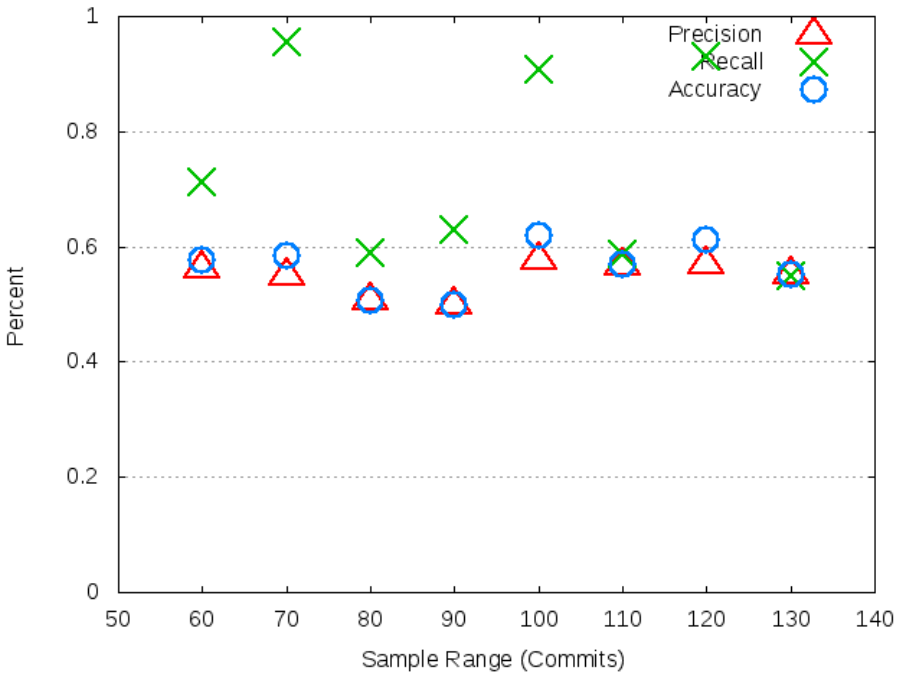


Figure A.36: SWR for deeplearning4j using RF

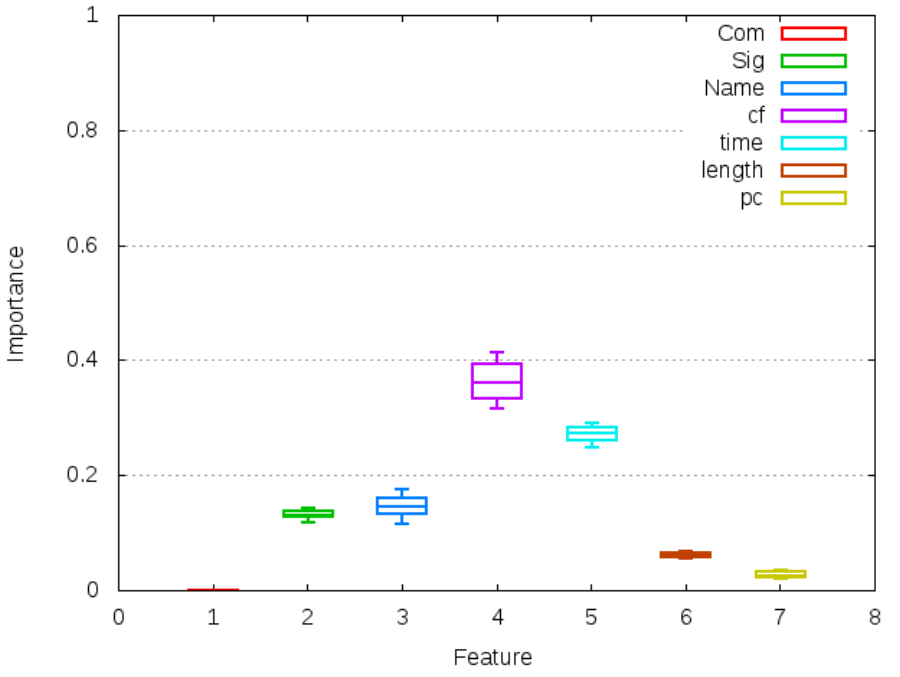


Figure A.37: Feature Importance SWR for deeplearning4j using RF

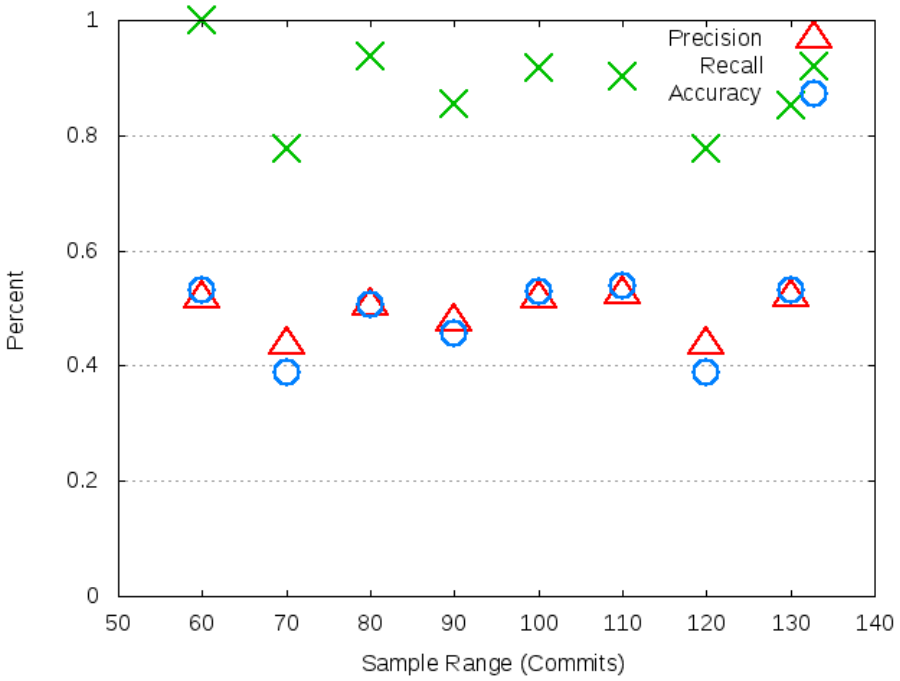


Figure A.38: SWR for fresco using RF

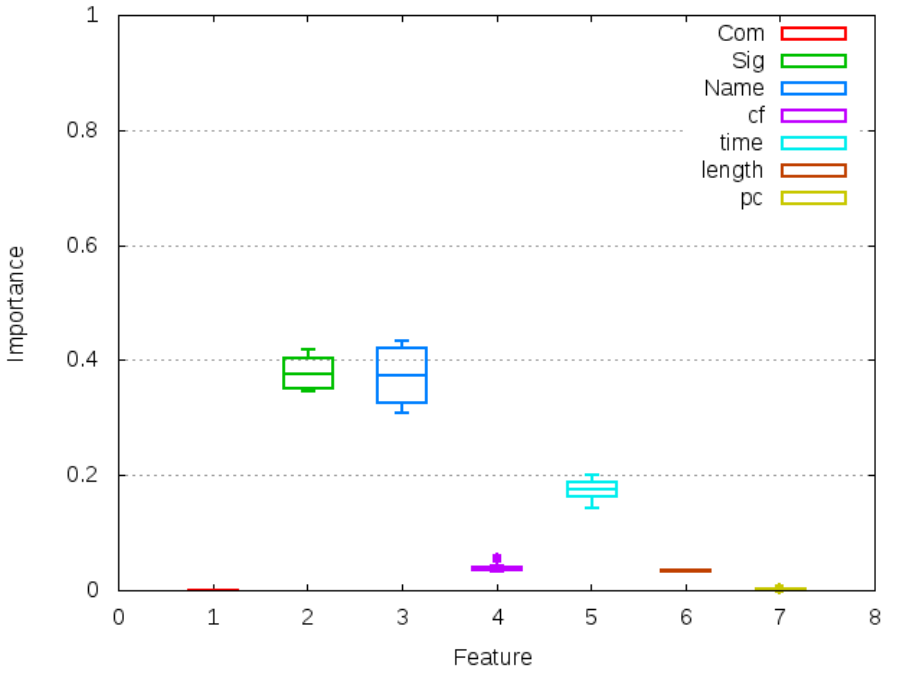


Figure A.39: Feature Importance SWR for fresco using RF

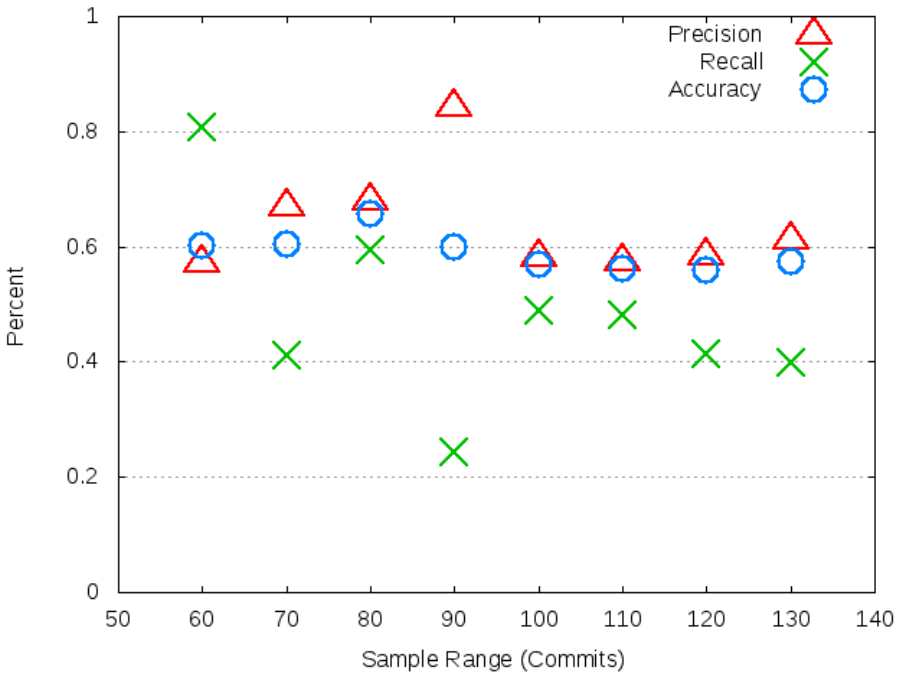


Figure A.40: SWR for governator using RF

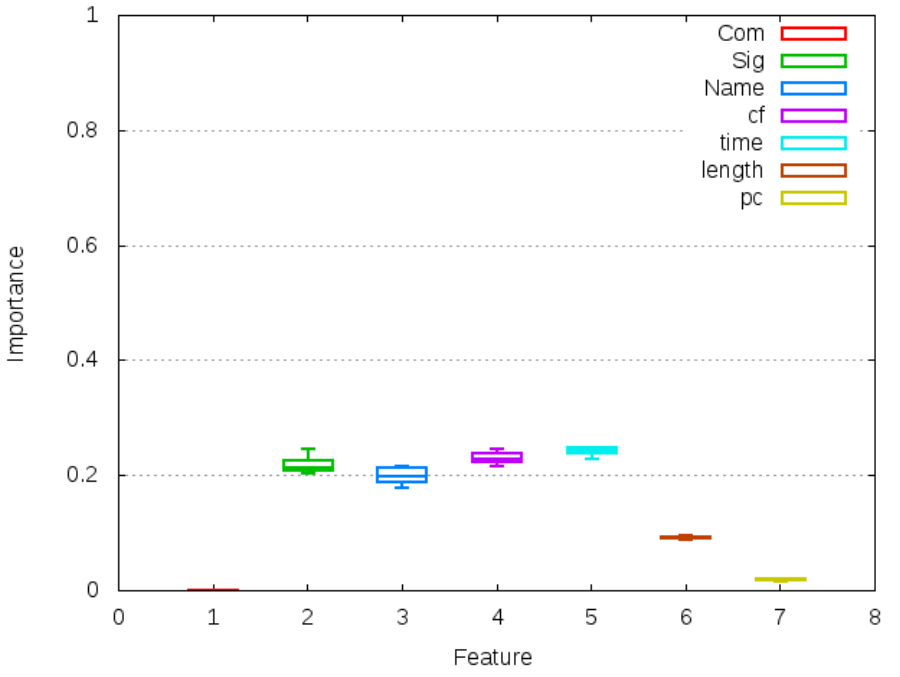


Figure A.41: Feature Importance SWR for governator using RF

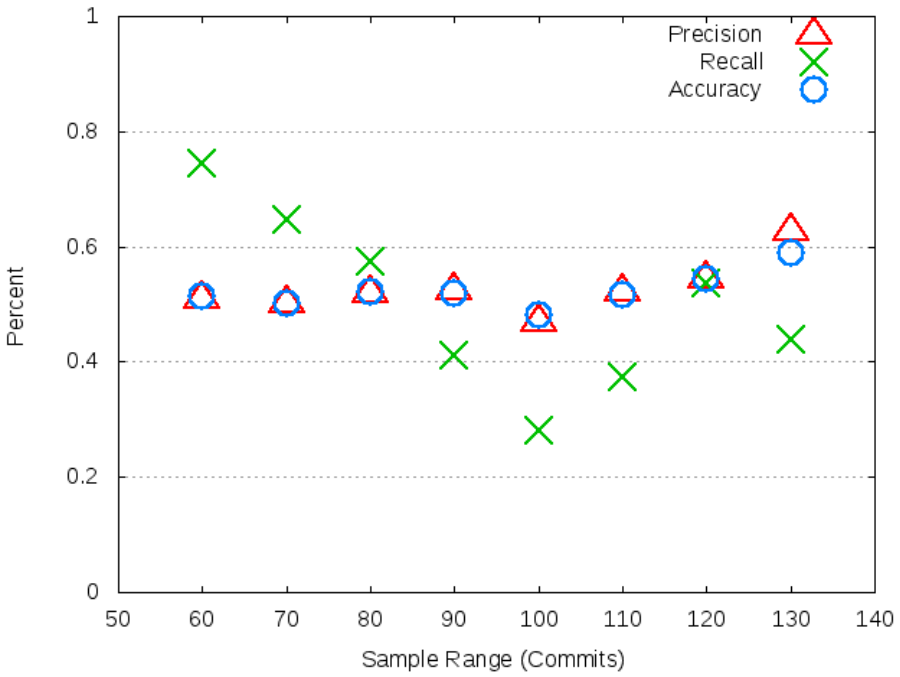


Figure A.42: SWR for greenDAO using RF

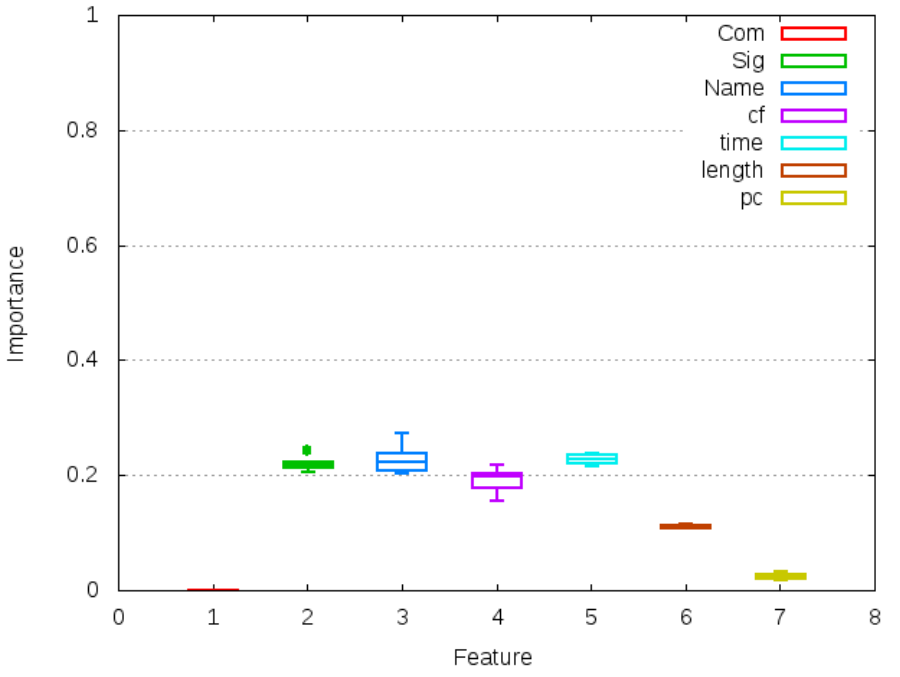


Figure A.43: Feature Importance SWR for greenDAO using RF

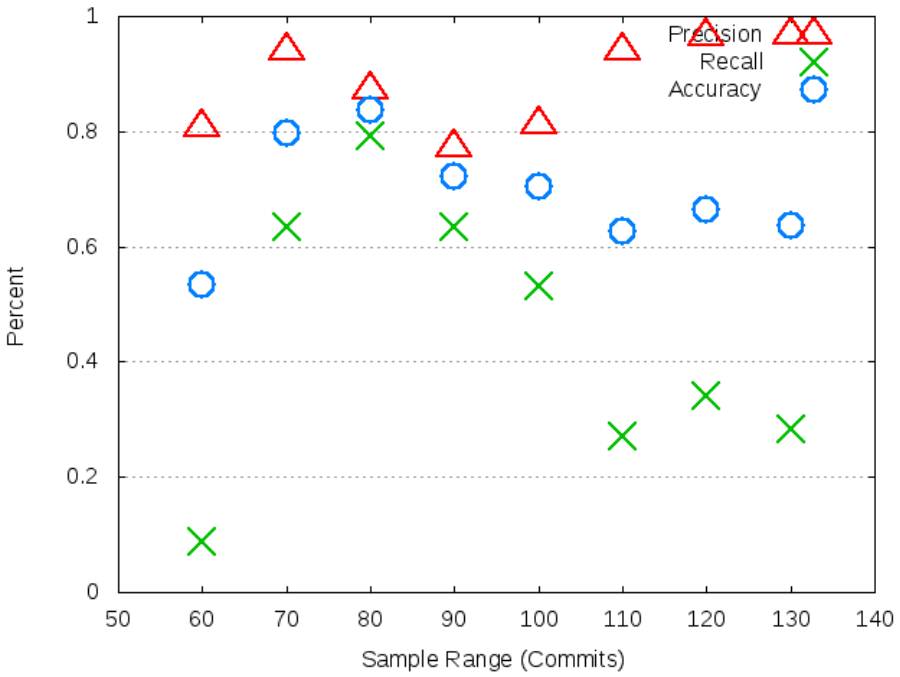


Figure A.44: SWR for http-request using RF

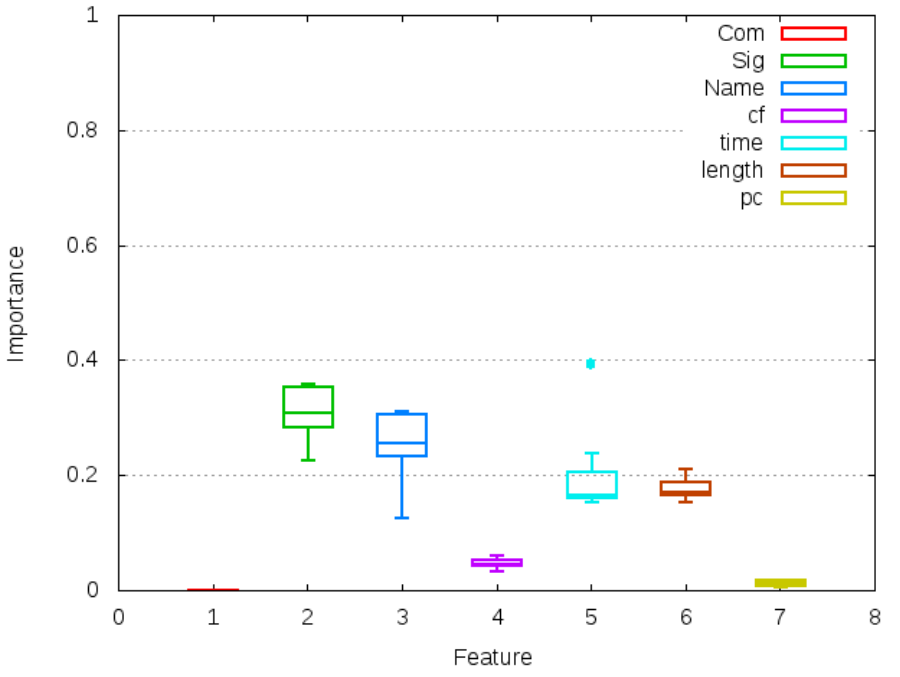


Figure A.45: Feature Importance SWR for http-request using RF

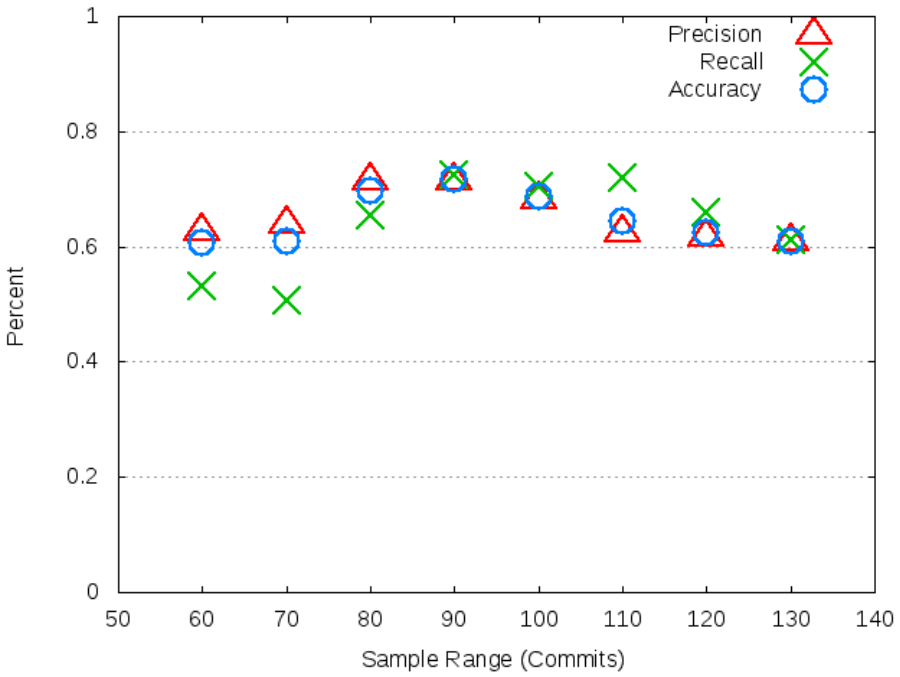


Figure A.46: SWR for ion using RF

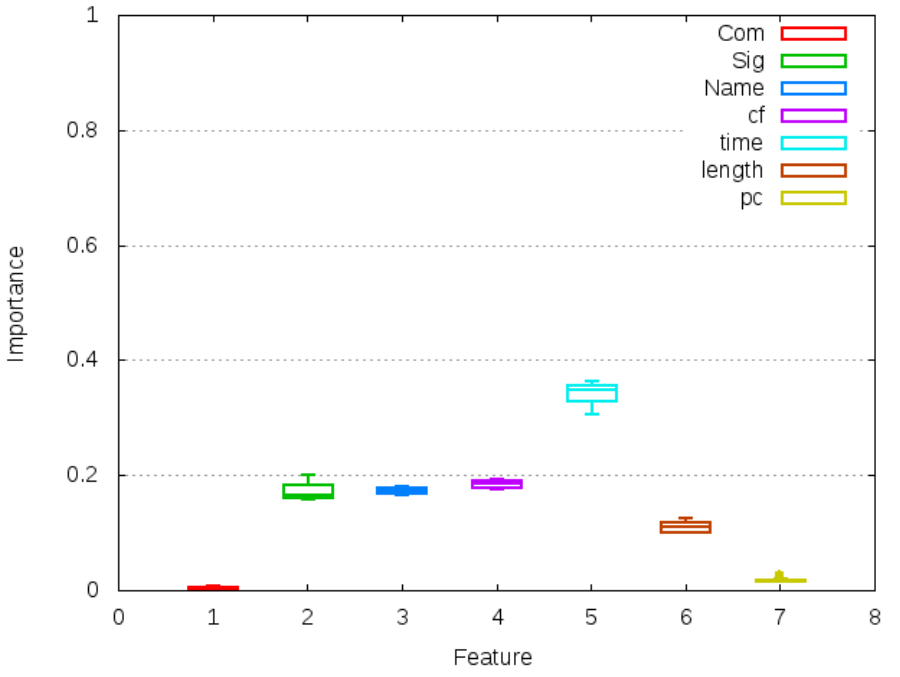


Figure A.47: Feature Importance SWR for ion using RF

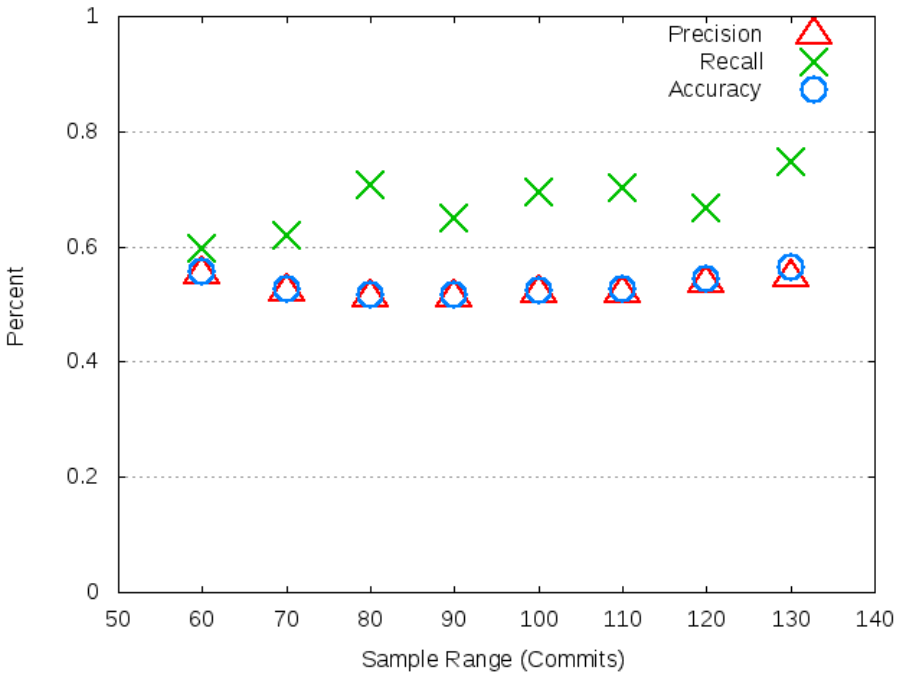


Figure A.48: SWR for jadx using RF

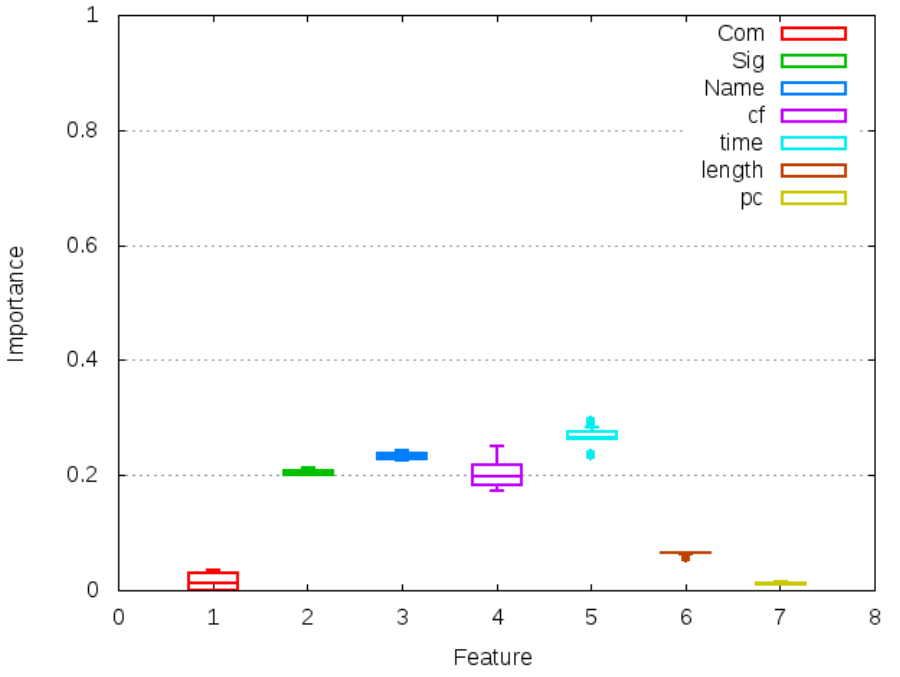


Figure A.49: Feature Importance SWR for jadx using RF

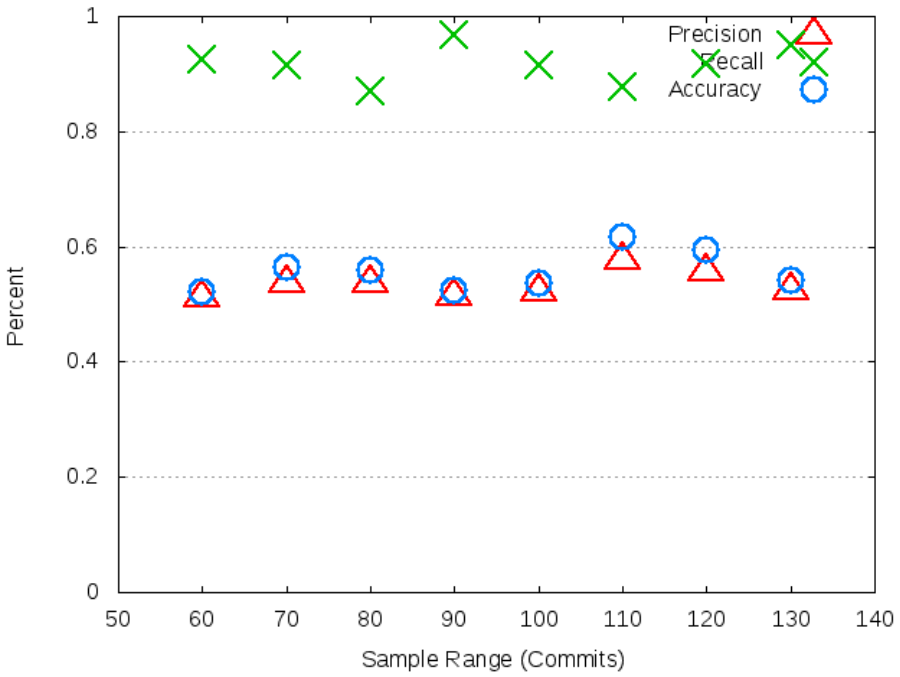


Figure A.50: SWR for mapstruct using RF

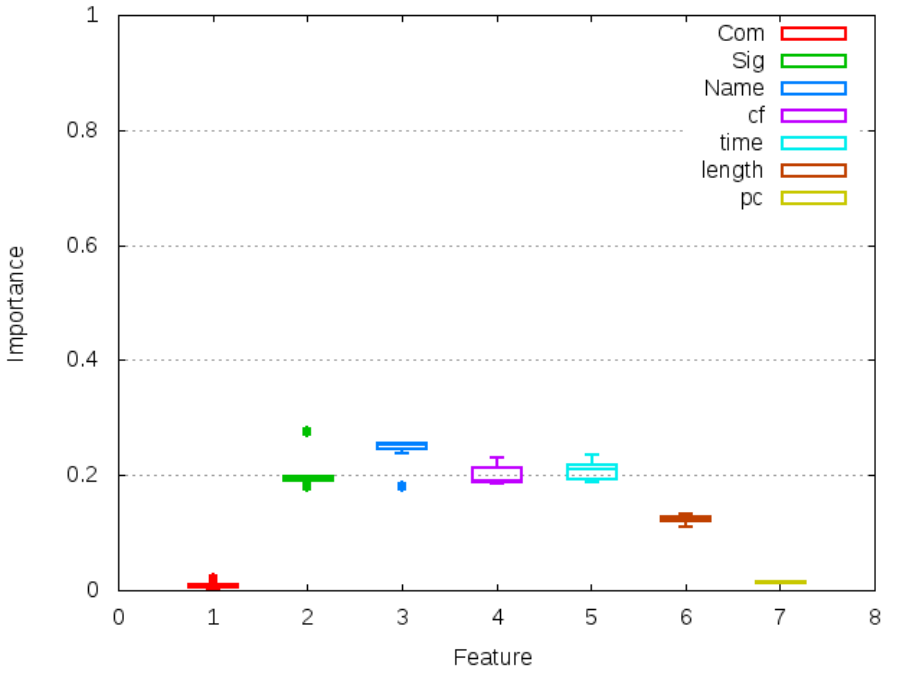


Figure A.51: Feature Importance SWR for mapstruct using RF

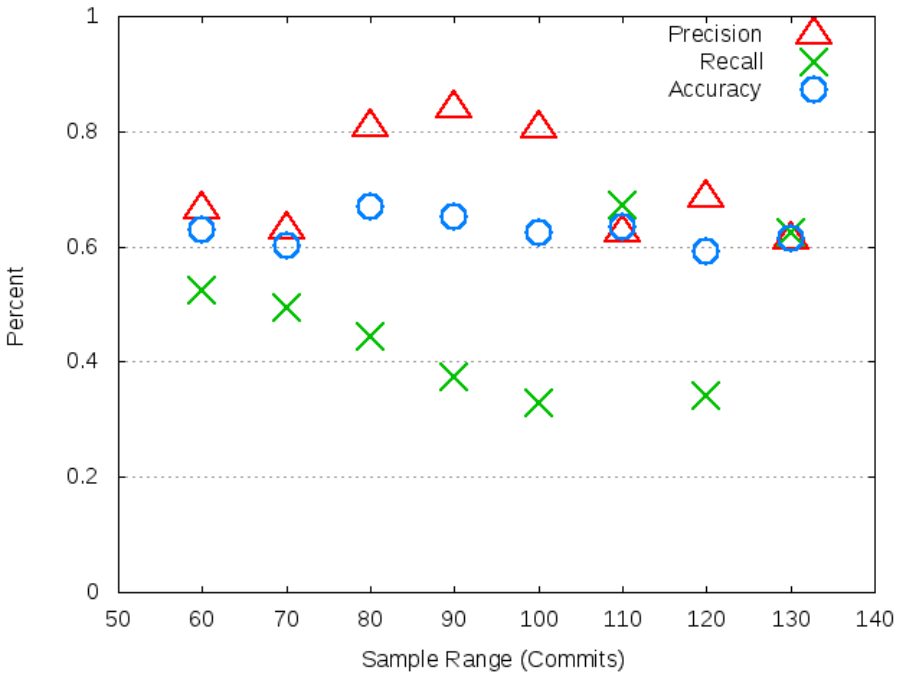


Figure A.52: SWR for nettosphere using RF

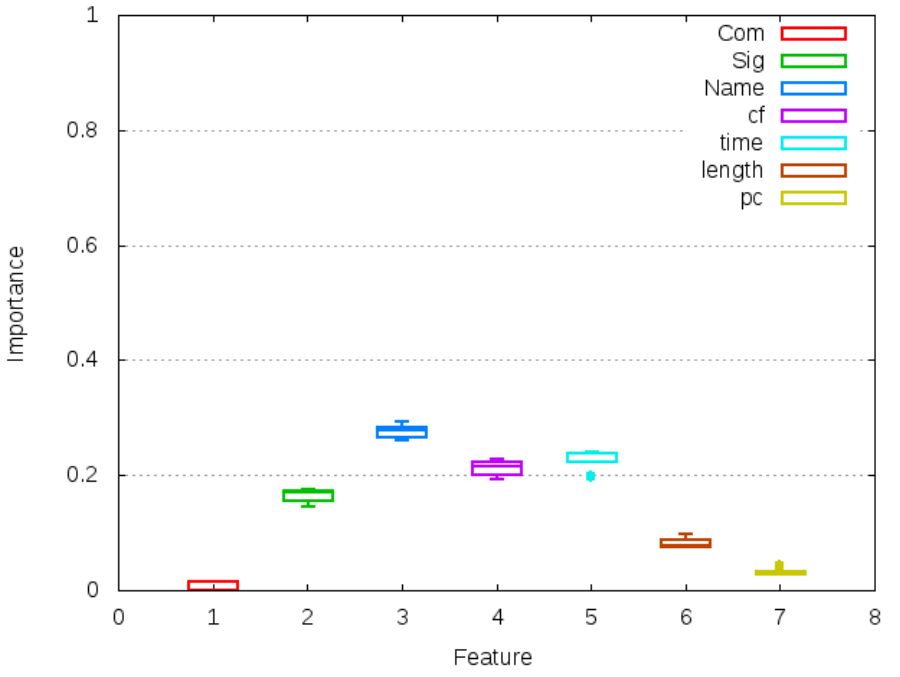


Figure A.53: Feature Importance SWR for nettosphere using RF

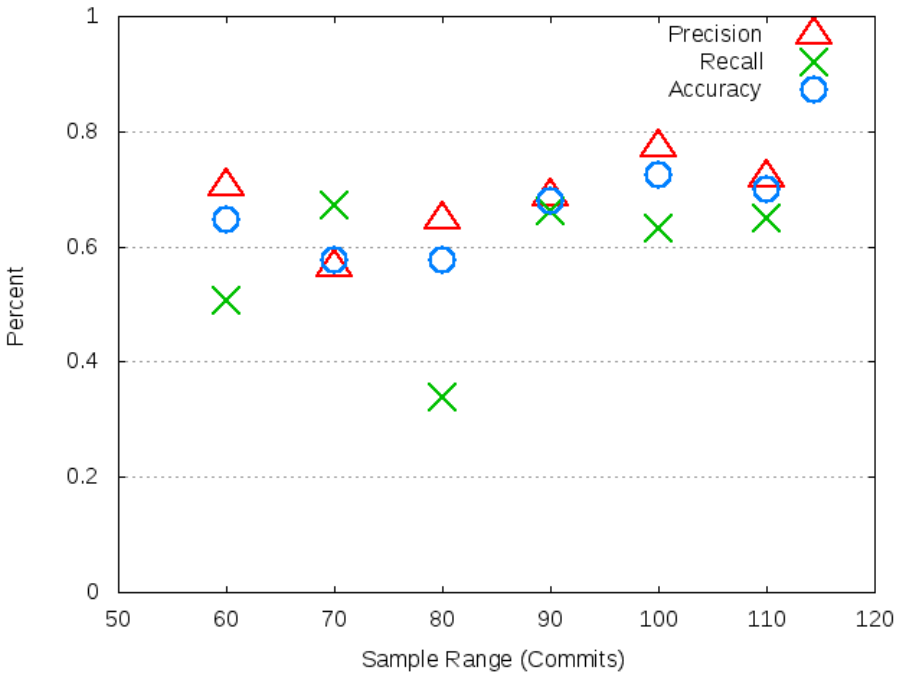


Figure A.54: SWR for parceler using RF

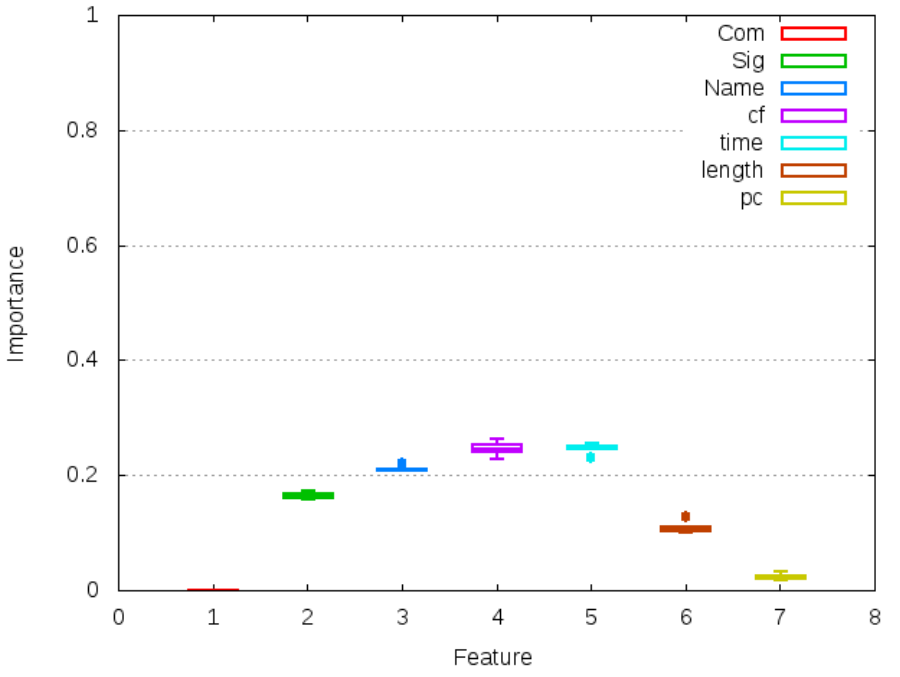


Figure A.55: Feature Importance SWR for parceler using RF

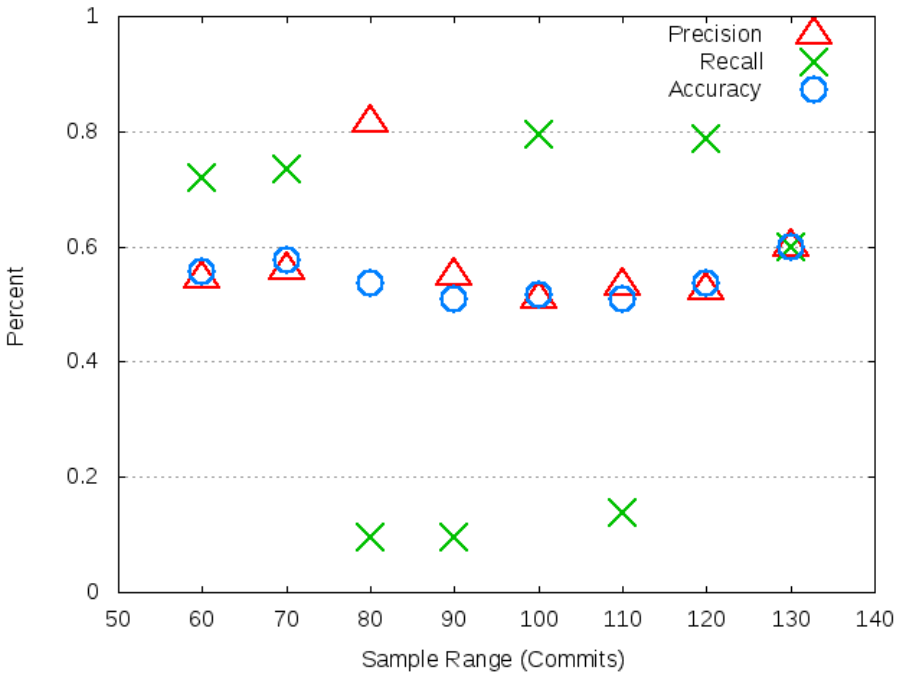


Figure A.56: SWR for retrolambda using RF

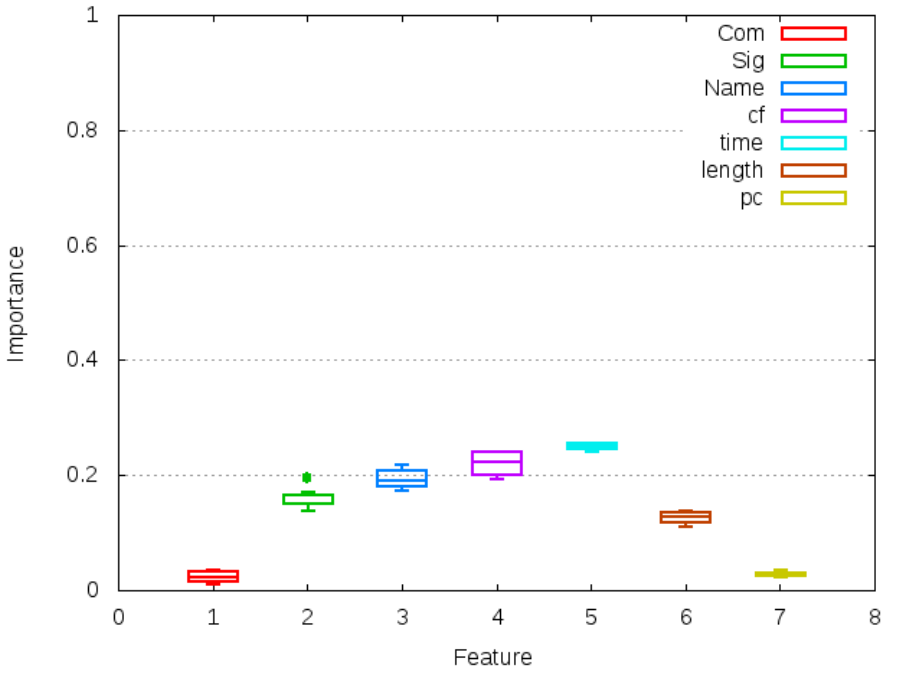


Figure A.57: Feature Importance SWR for retrolambda using RF

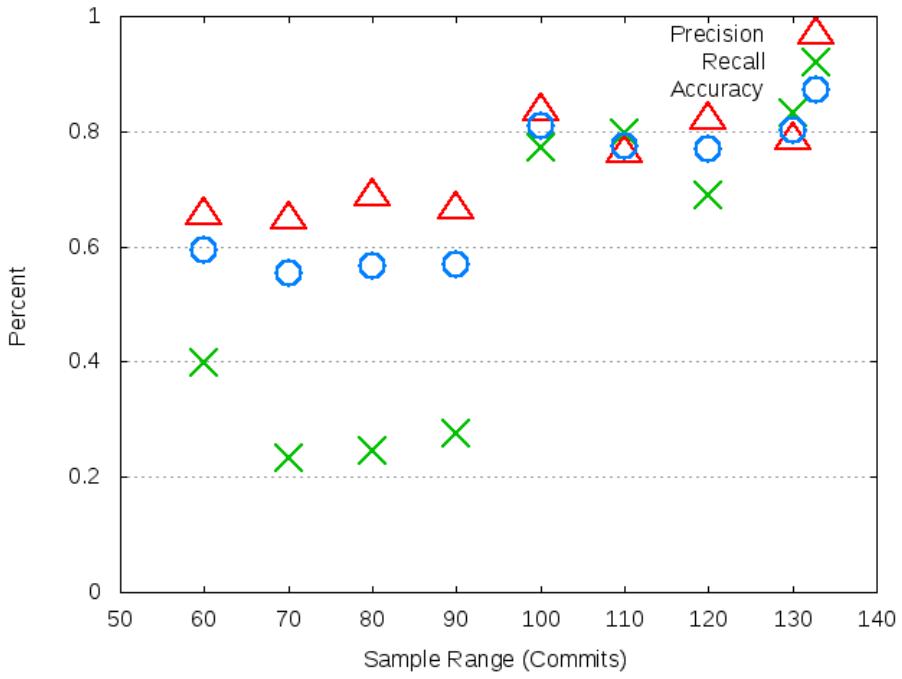


Figure A.58: SWR for ShowcaseView using RF

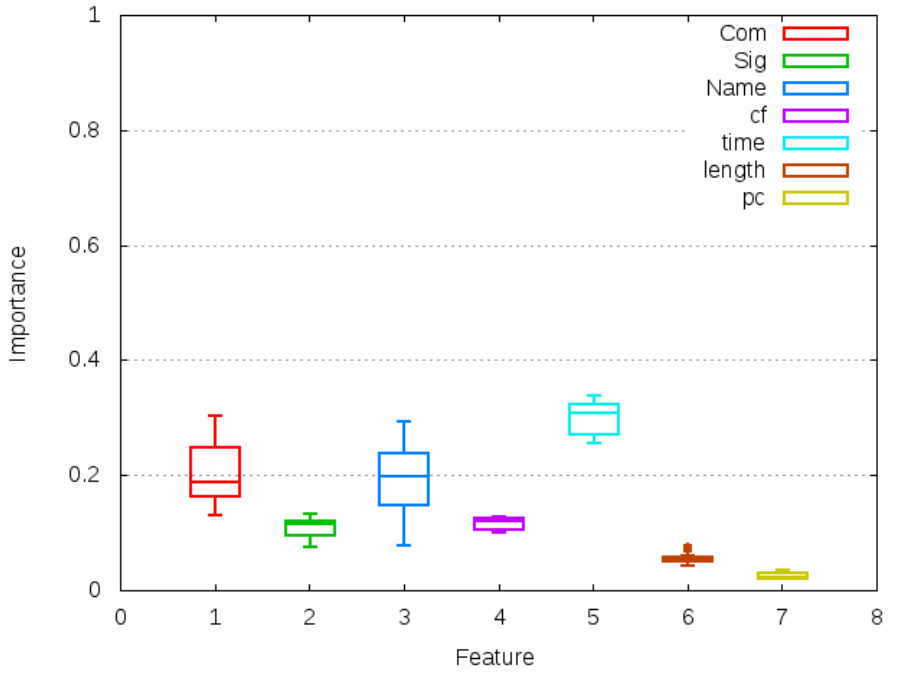


Figure A.59: Feature Importance SWR for ShowcaseView using RF

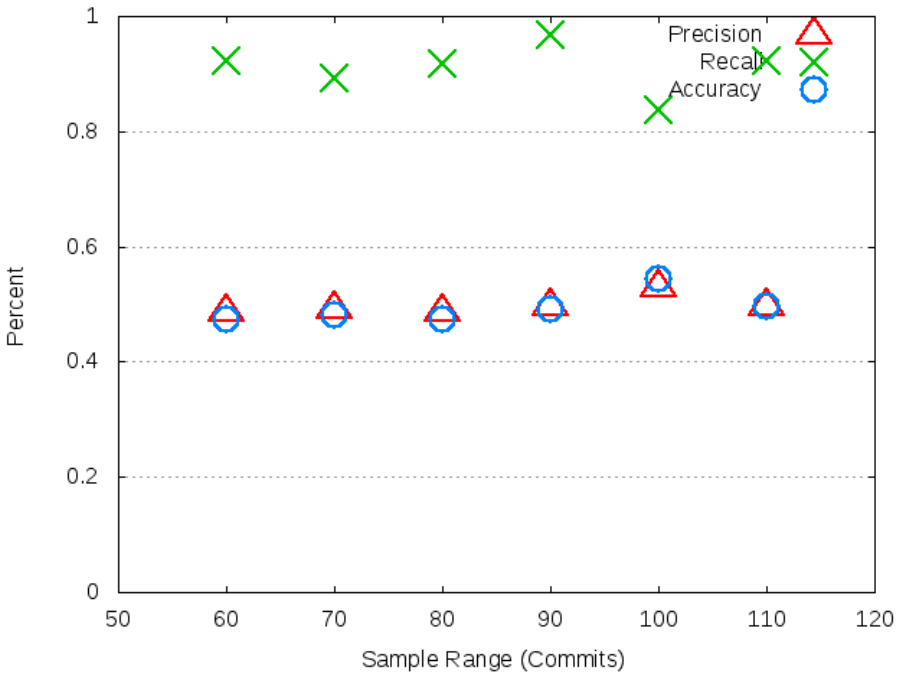


Figure A.60: SWR for smile using RF

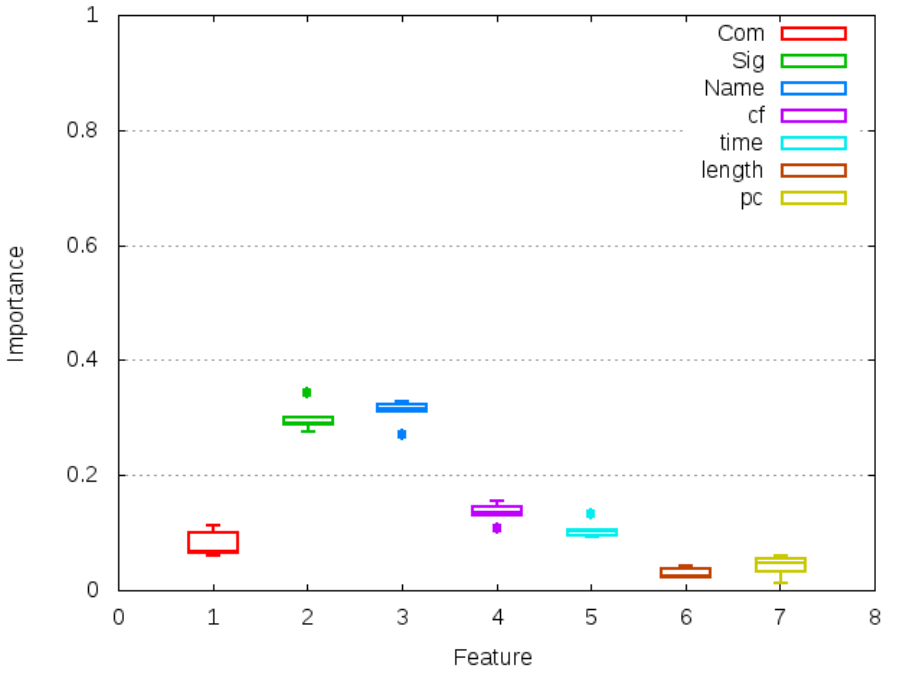


Figure A.61: Feature Importance SWR for smile using RF

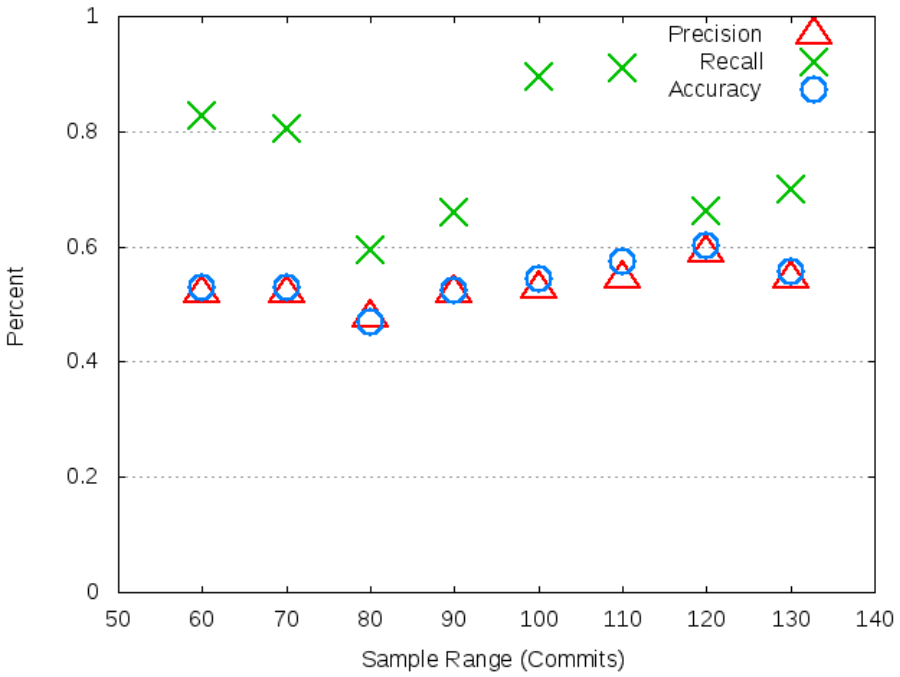


Figure A.62: SWR for spark using RF

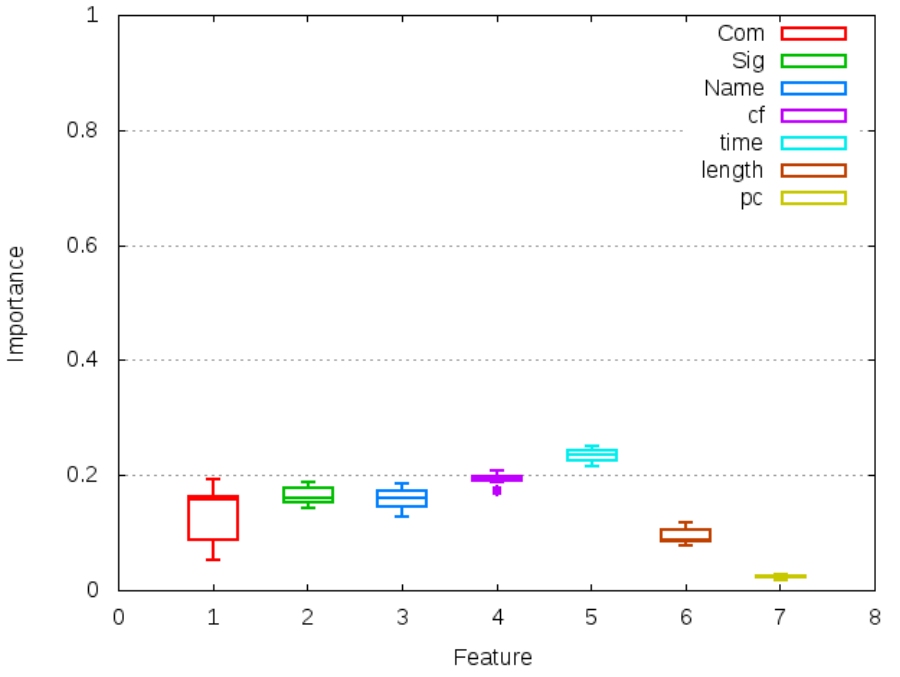


Figure A.63: Feature Importance SWR for spark using RF

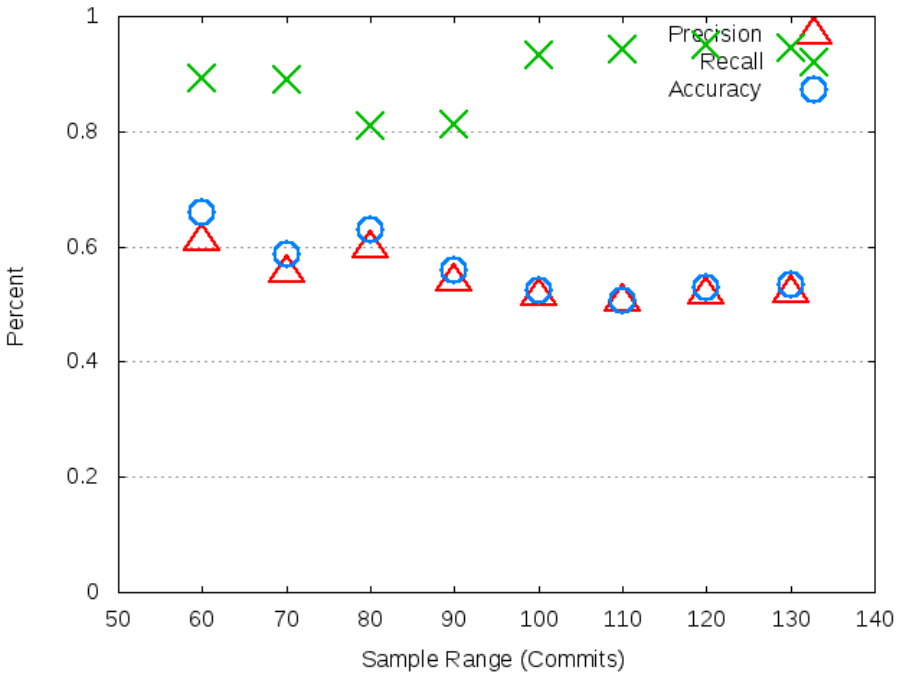


Figure A.64: SWR for storm using RF

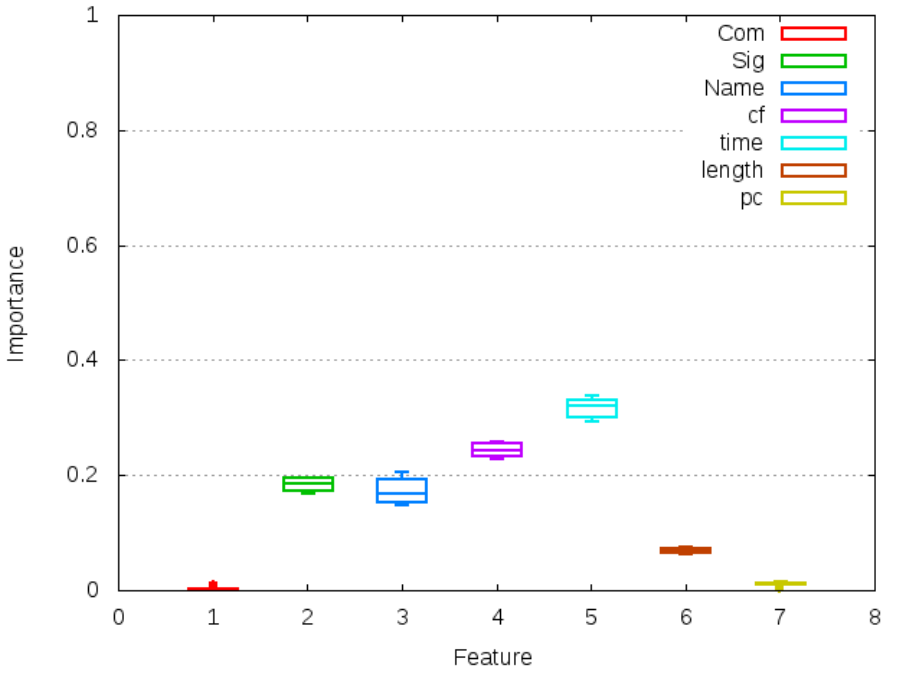


Figure A.65: Feature Importance SWR for storm using RF

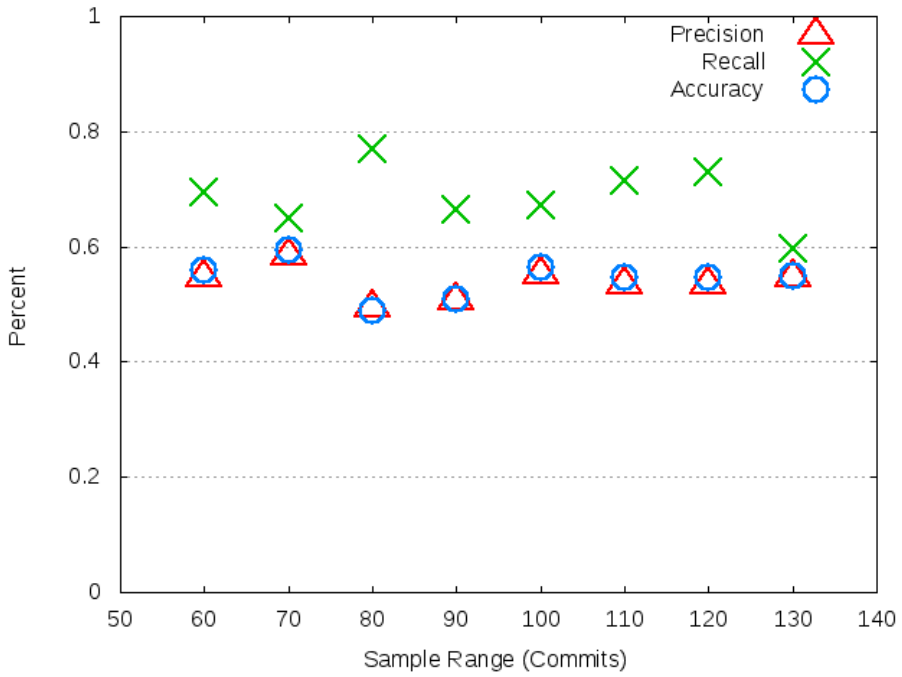


Figure A.66: SWR for tempto using RF

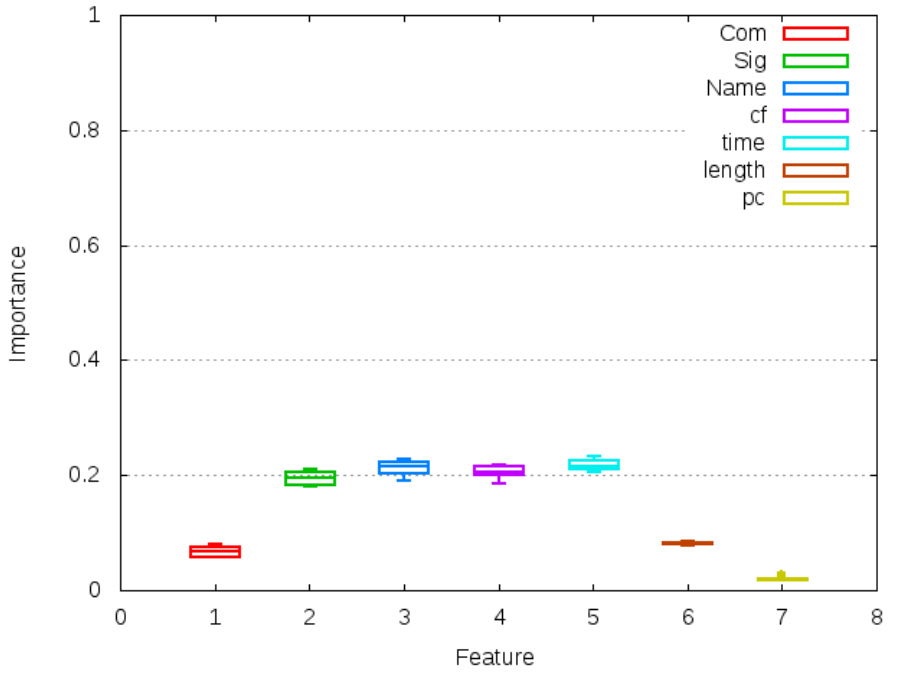


Figure A.67: Feature Importance SWR for tempto using RF

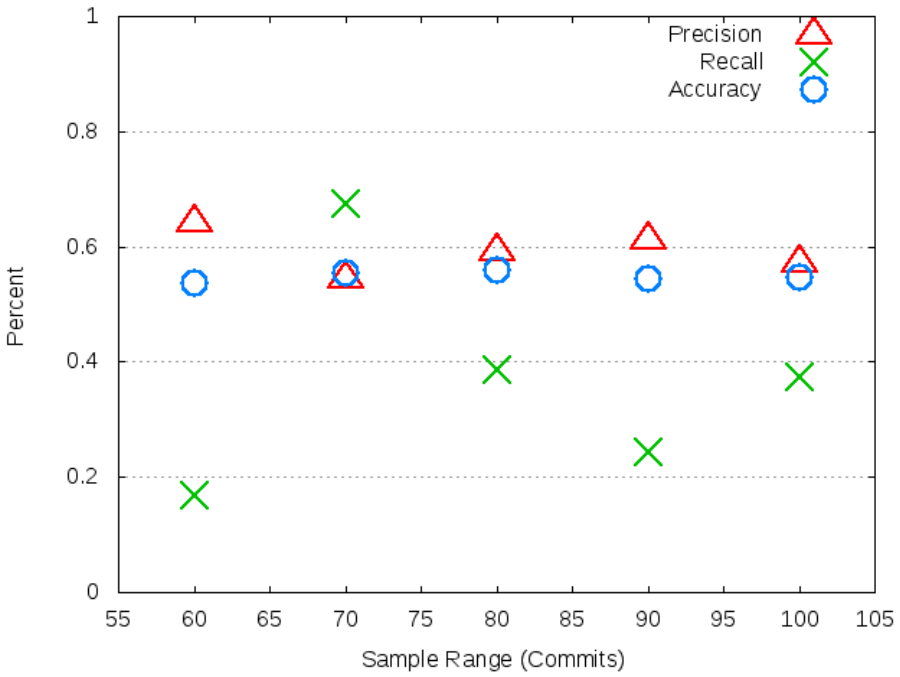


Figure A.68: SWR for yardstick using RF

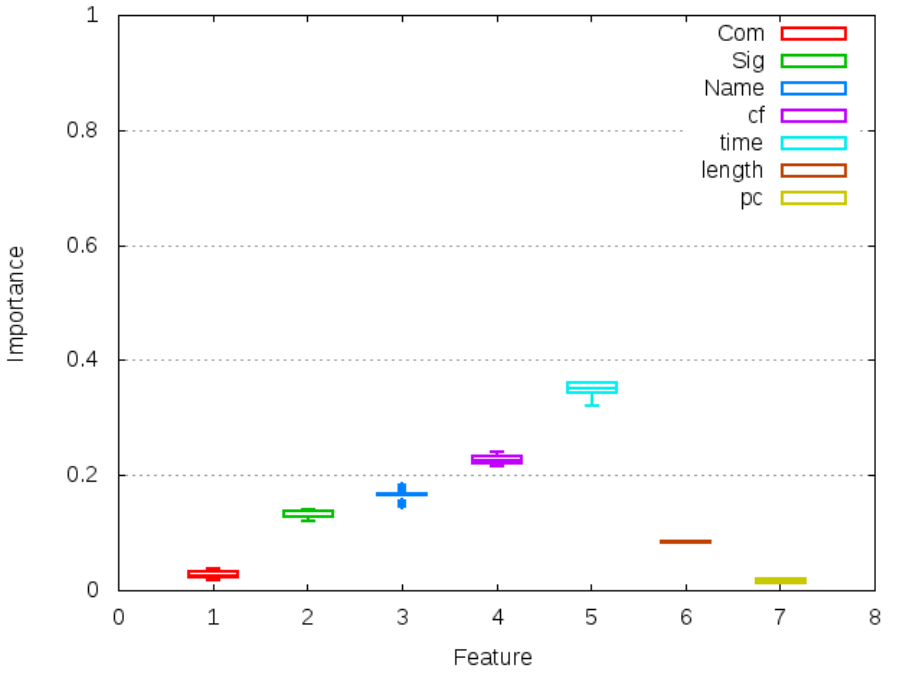


Figure A.69: Feature Importance SWR for yardstick using RF

A.2 Experiment 2

A.2.1 Support Vector Machine

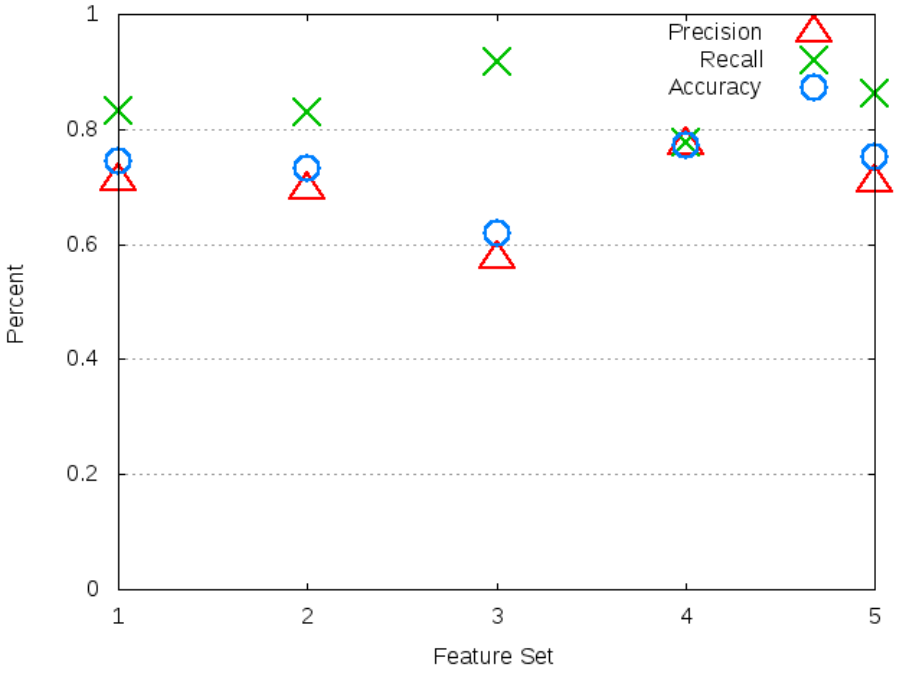


Figure A.70: Feature for acra using SVM

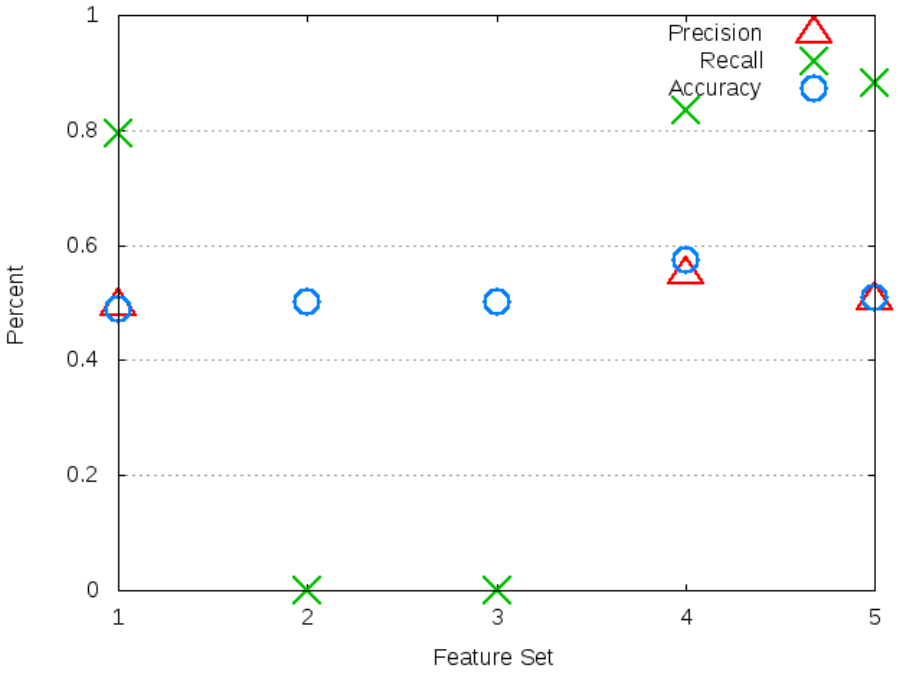


Figure A.71: Feature for arquillian-core using SVM

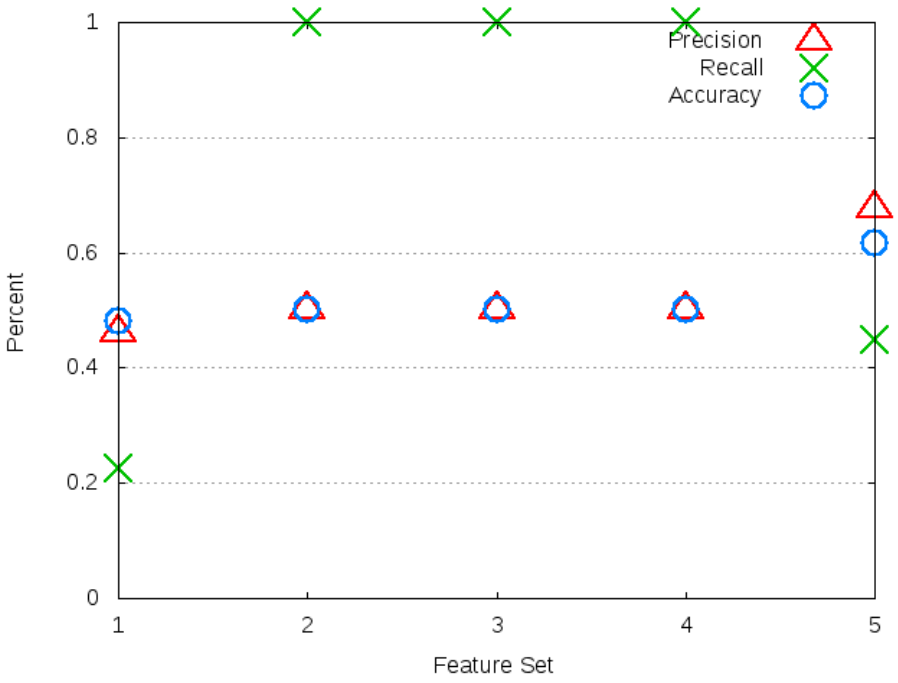


Figure A.72: Feature for blockly-android using SVM

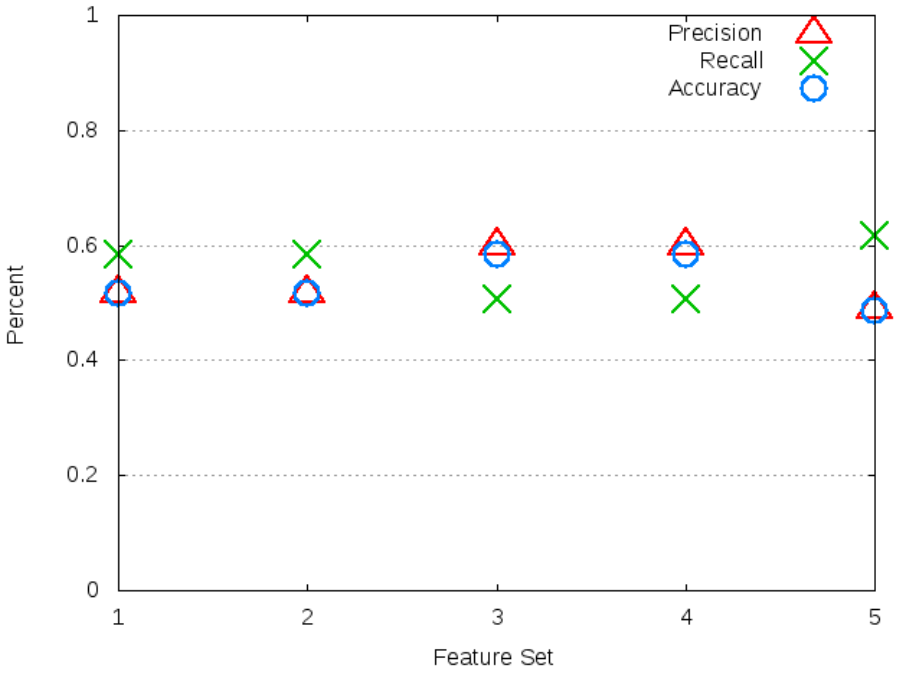


Figure A.73: Feature for brave using SVM

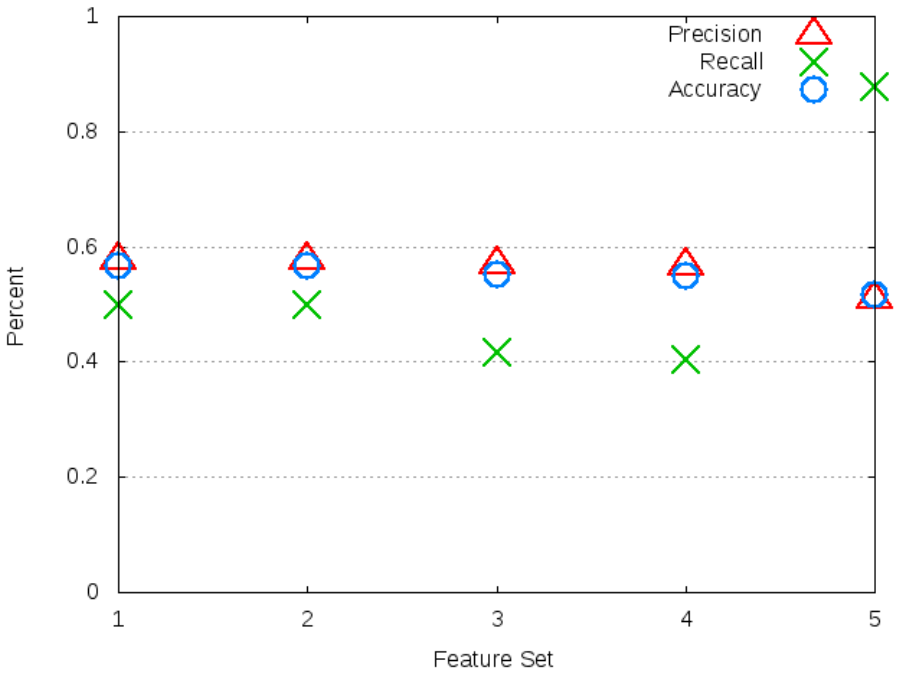


Figure A.74: Feature for cardslib using SVM

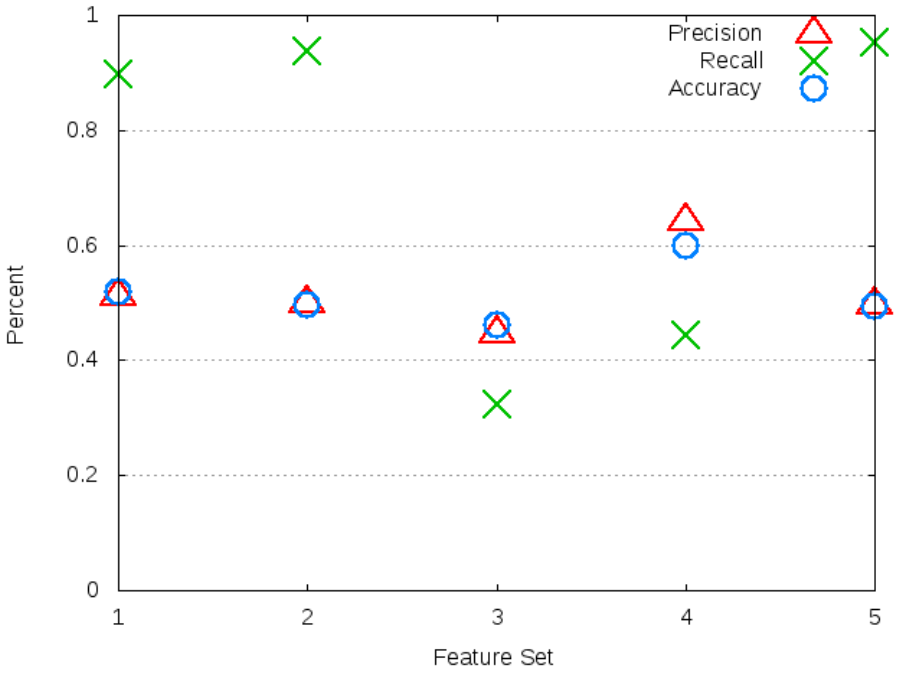


Figure A.75: Feature for dagger using SVM

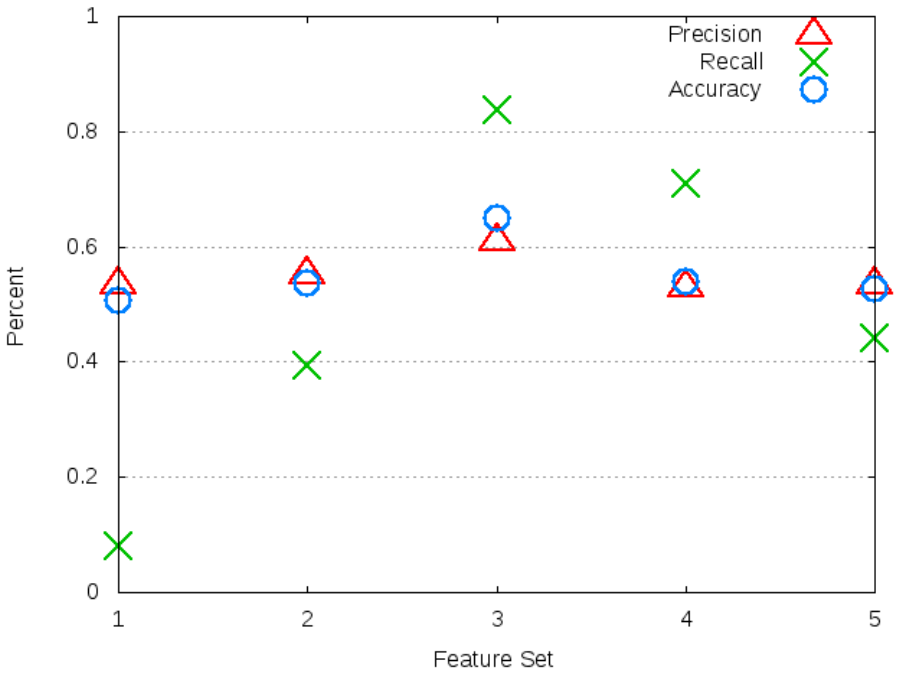


Figure A.76: Feature for deeplearning4j using SVM

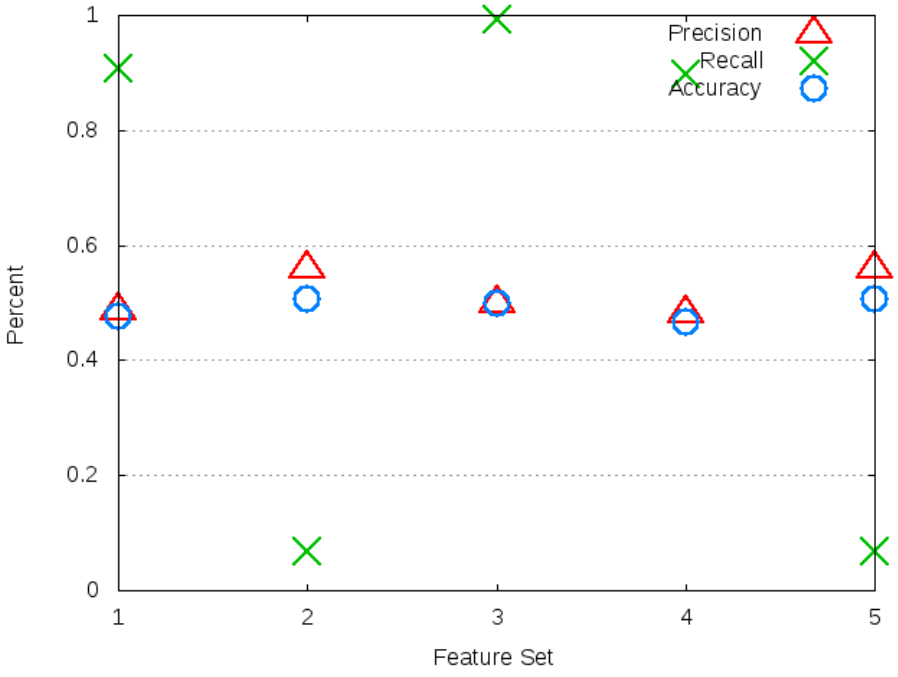


Figure A.77: Feature for fresco using SVM

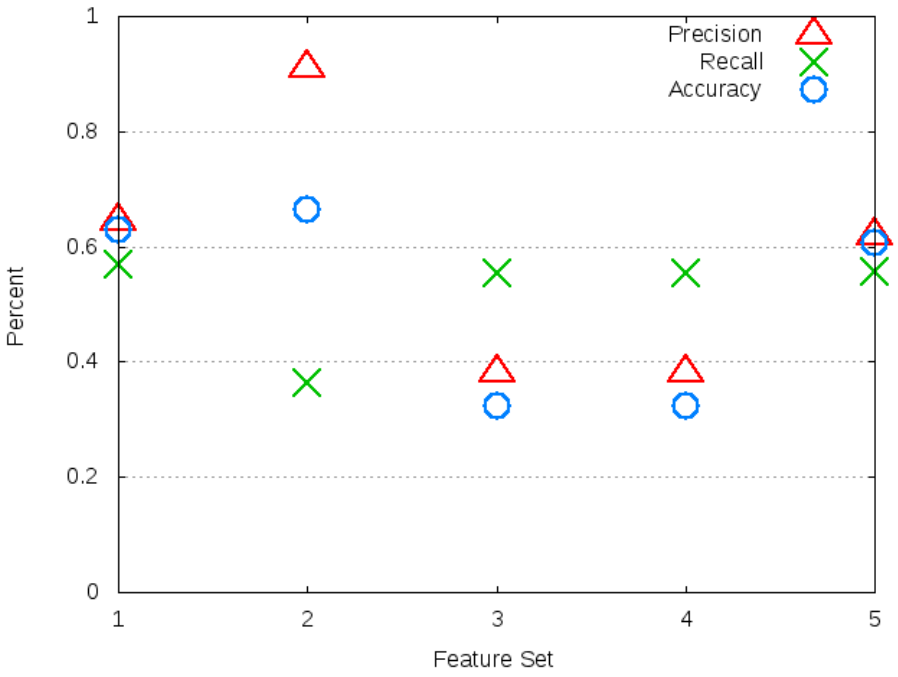


Figure A.78: Feature for governator using SVM

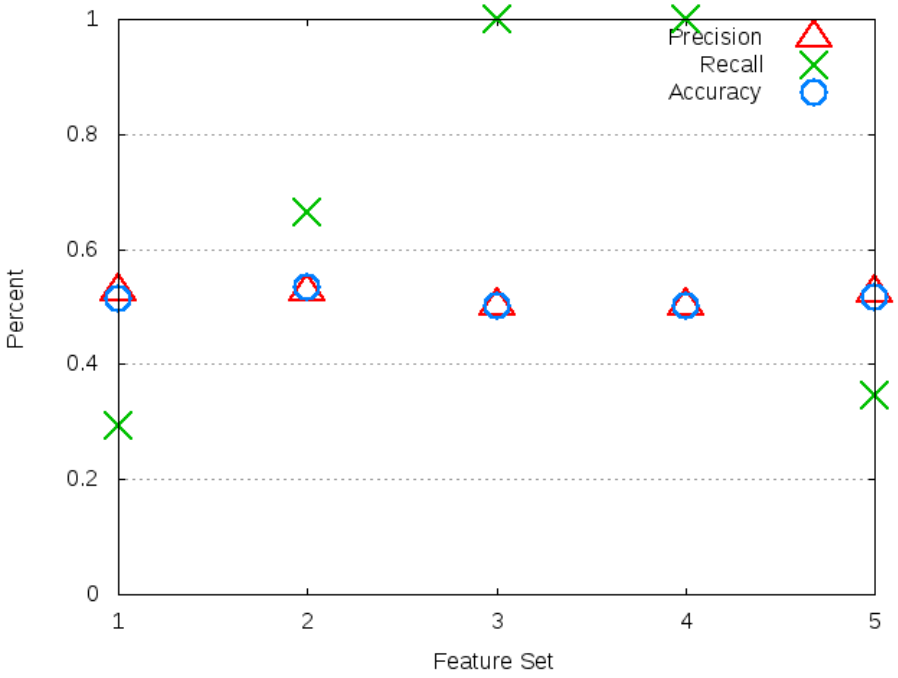


Figure A.79: Feature for greenDAO using SVM

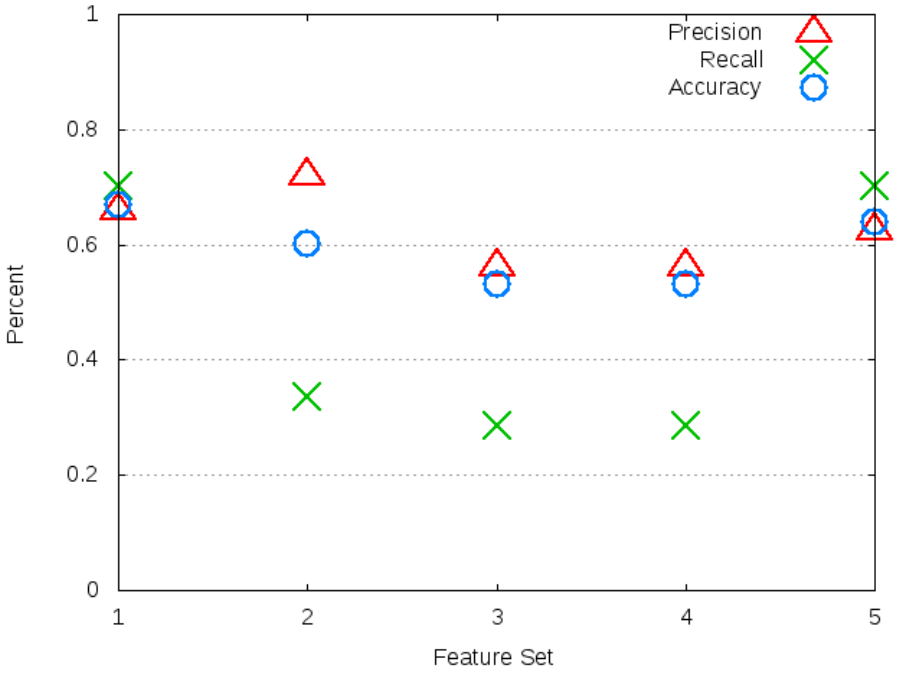


Figure A.80: Feature for http-request using SVM

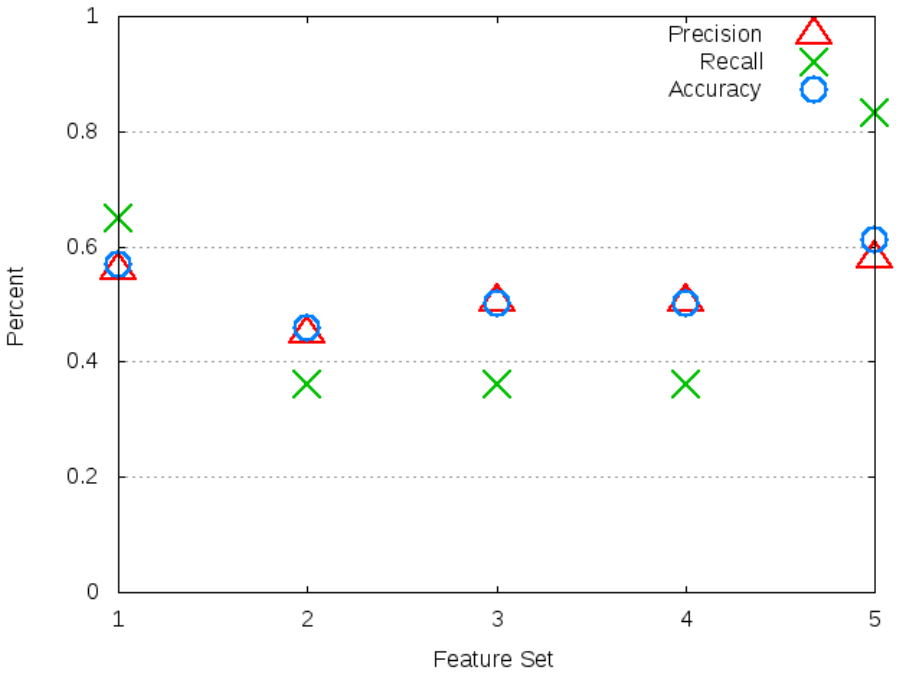


Figure A.81: Feature for ion using SVM

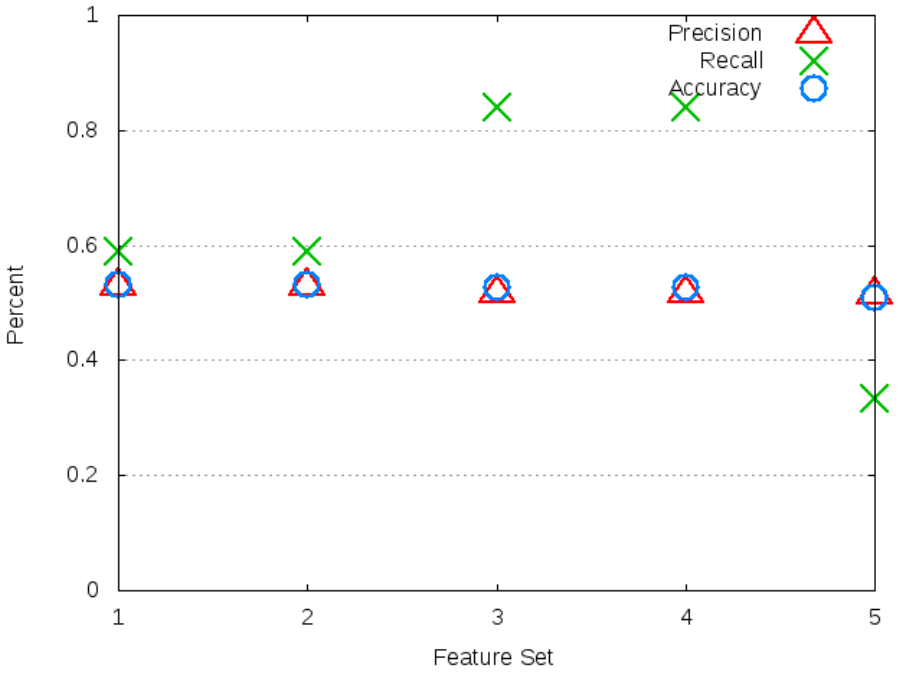


Figure A.82: Feature for jadx using SVM

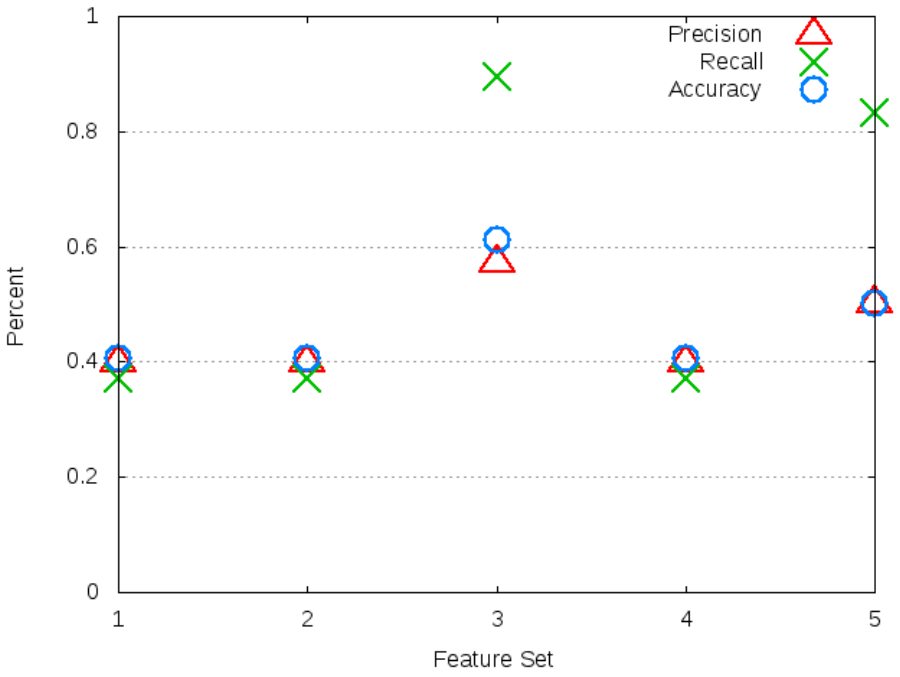


Figure A.83: Feature for mapstruct using SVM

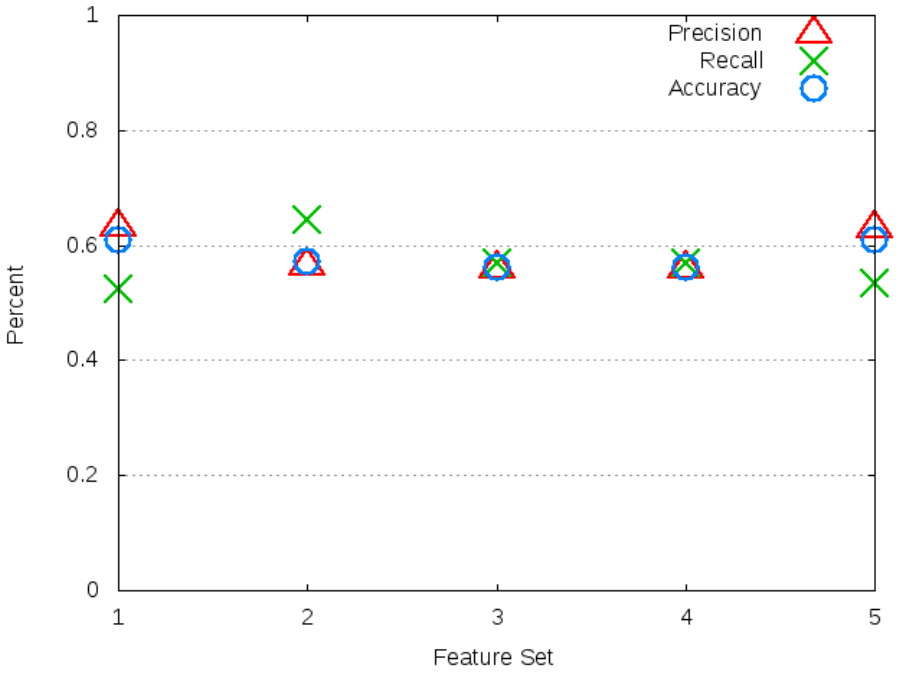


Figure A.84: Feature for nettosphere using SVM

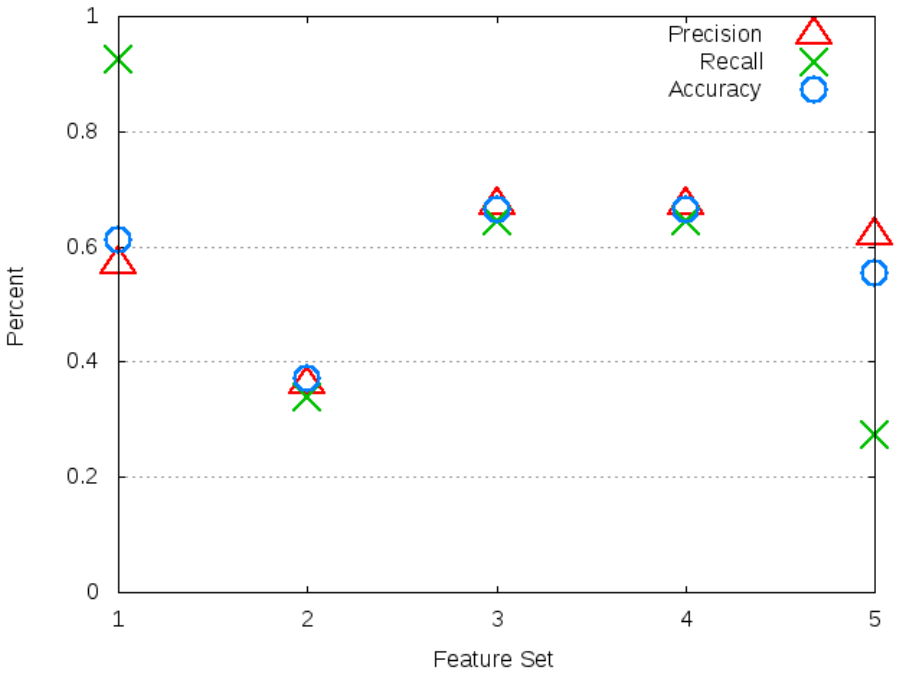


Figure A.85: Feature for parceler using SVM

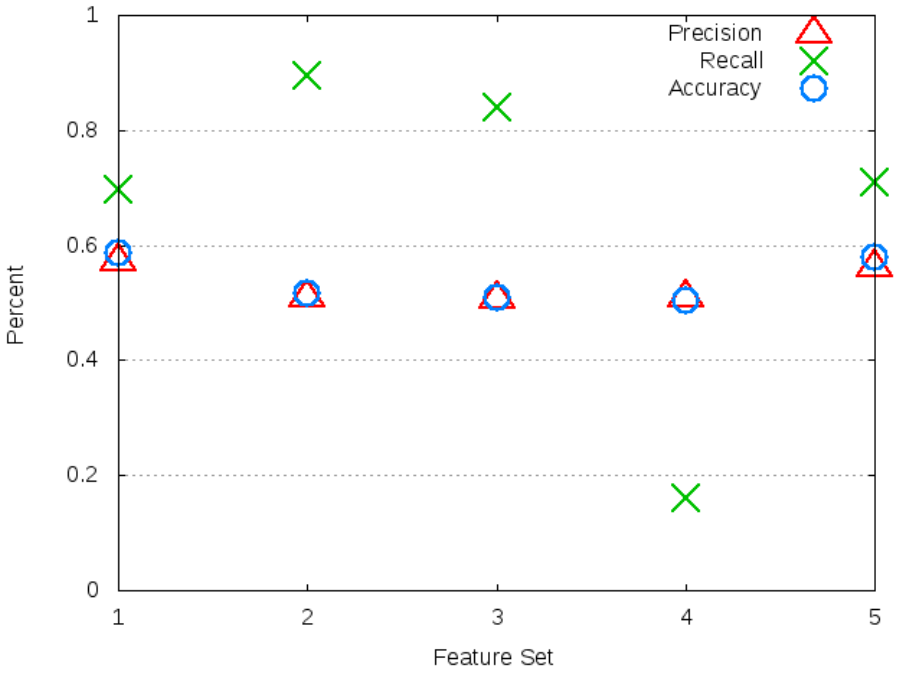


Figure A.86: Feature for retrolambda using SVM

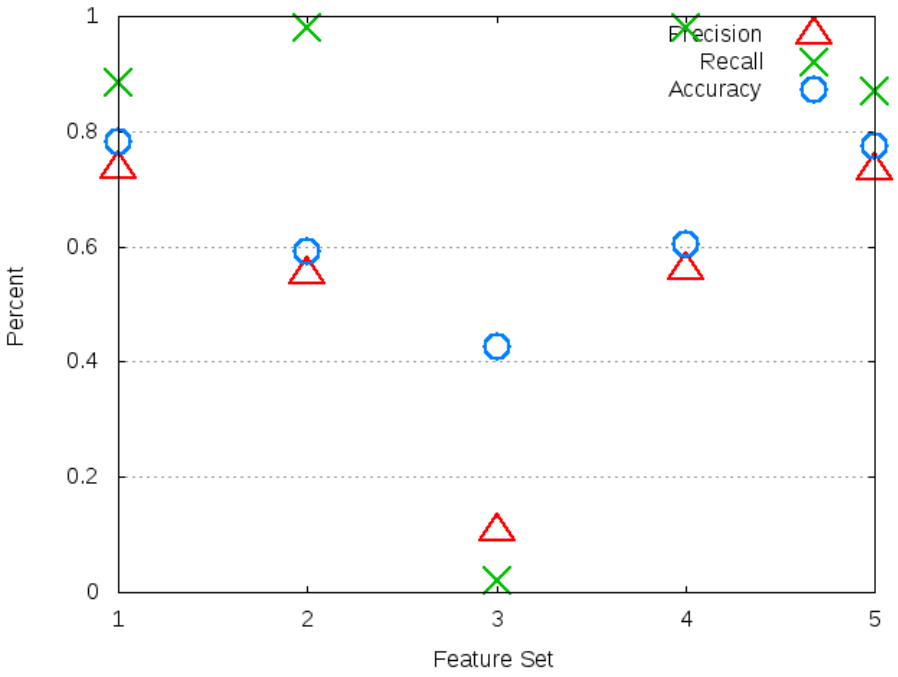


Figure A.87: Feature for ShowcaseView using SVM

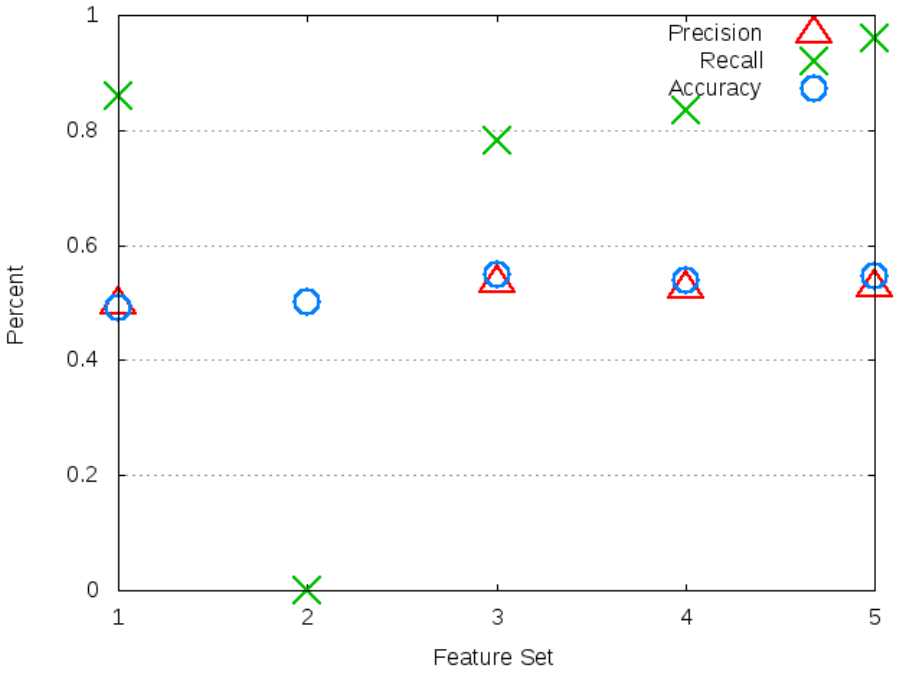


Figure A.88: Feature for smile using SVM

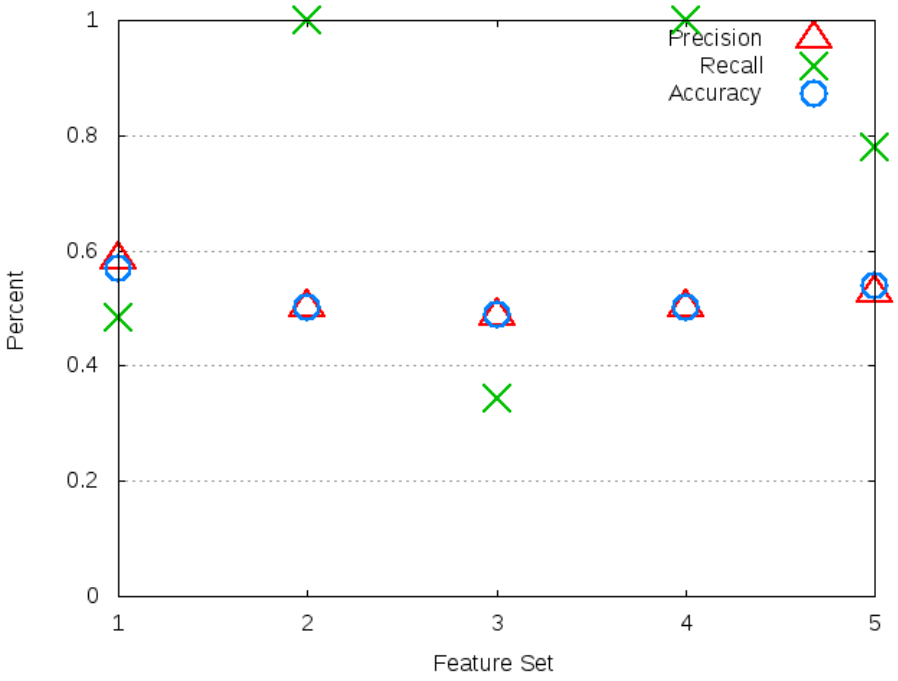


Figure A.89: Feature for spark using SVM

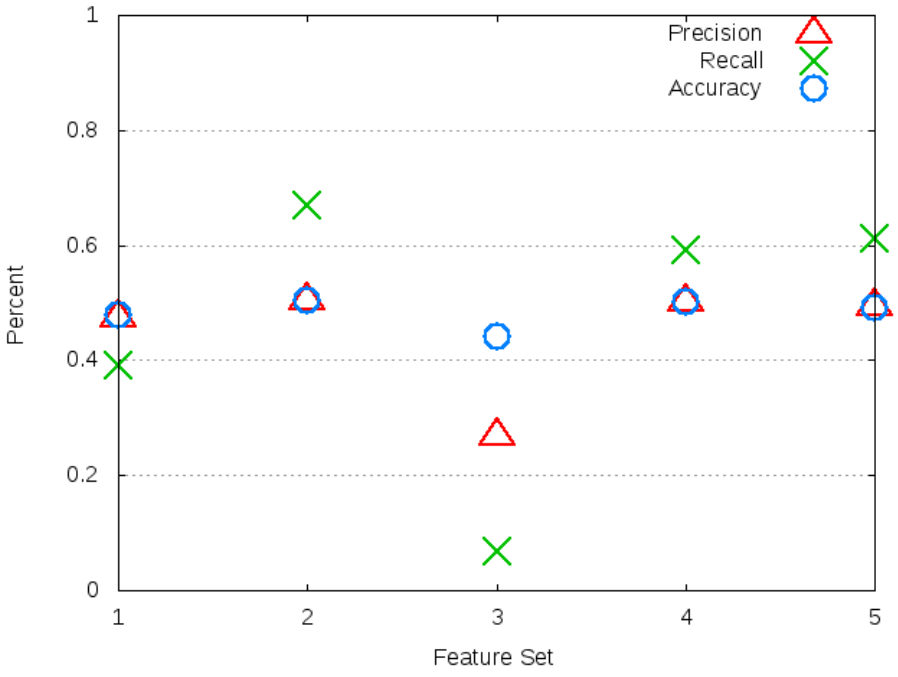


Figure A.90: Feature for storm using SVM

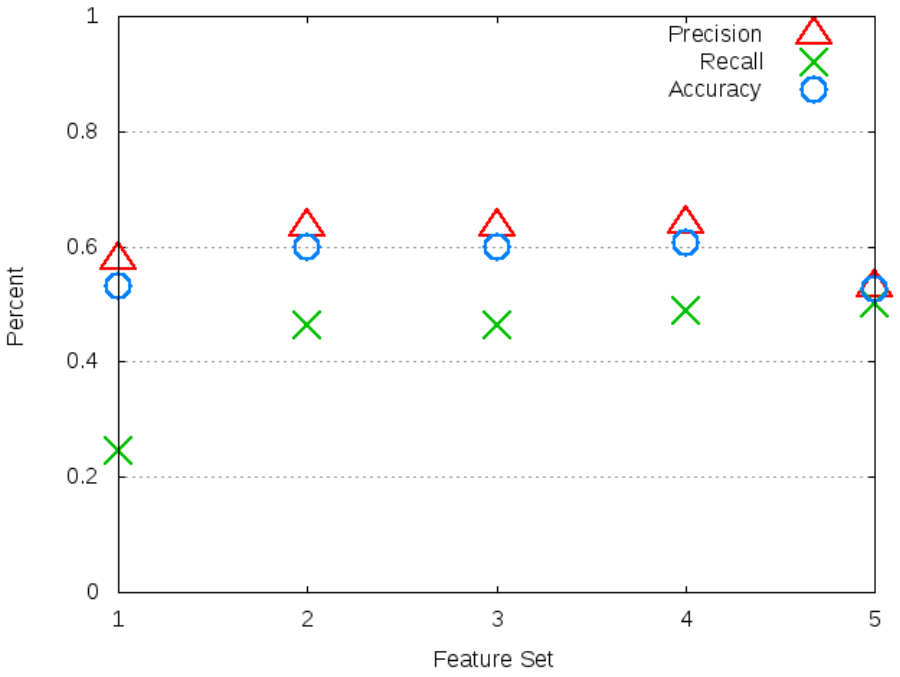


Figure A.91: Feature for tempto using SVM

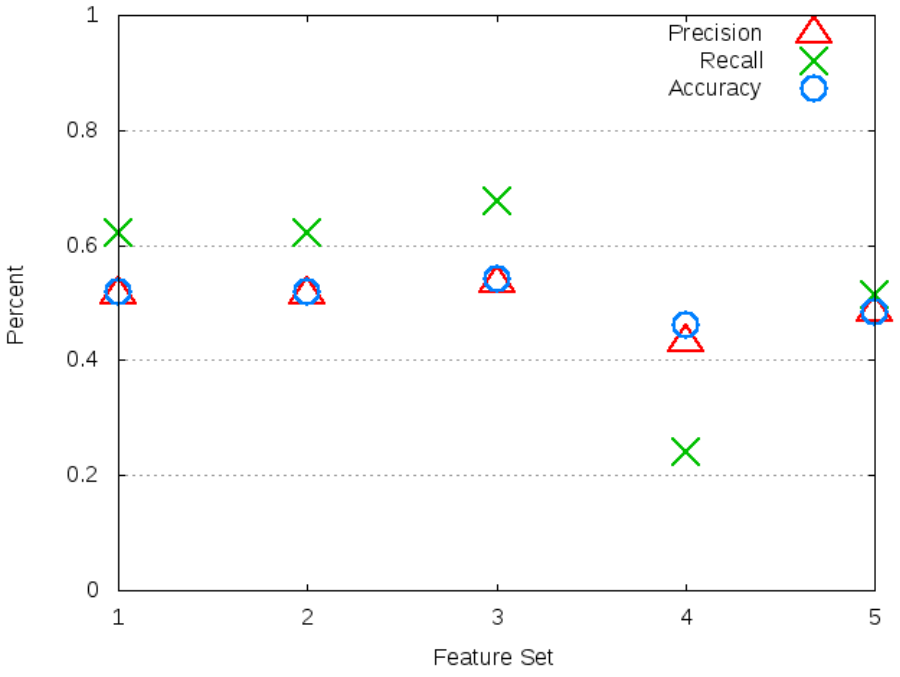


Figure A.92: Feature for yardstick using SVM

A.2.2 Random Forest

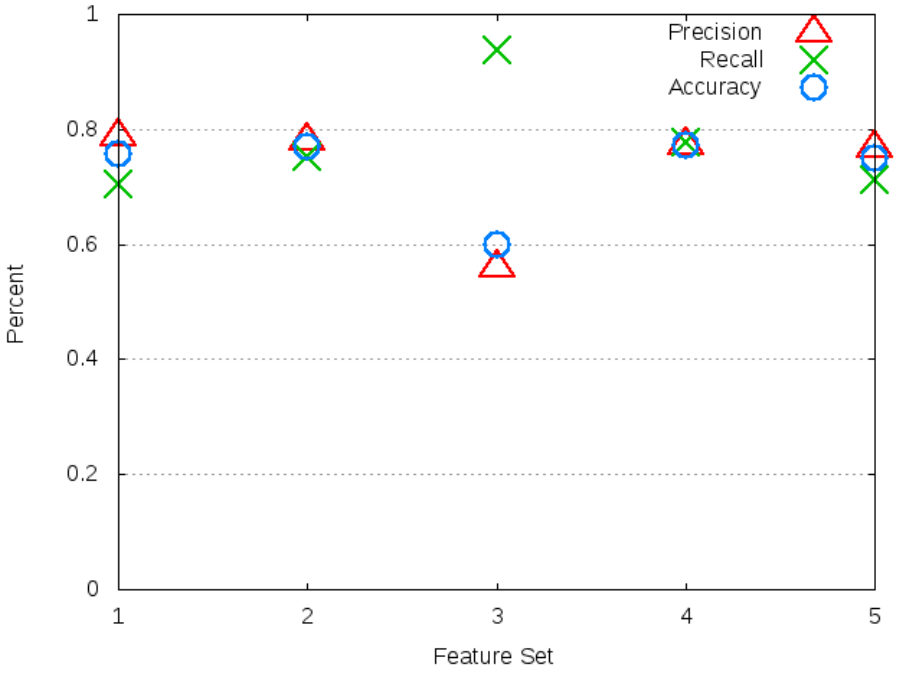


Figure A.93: Feature for acra using RF

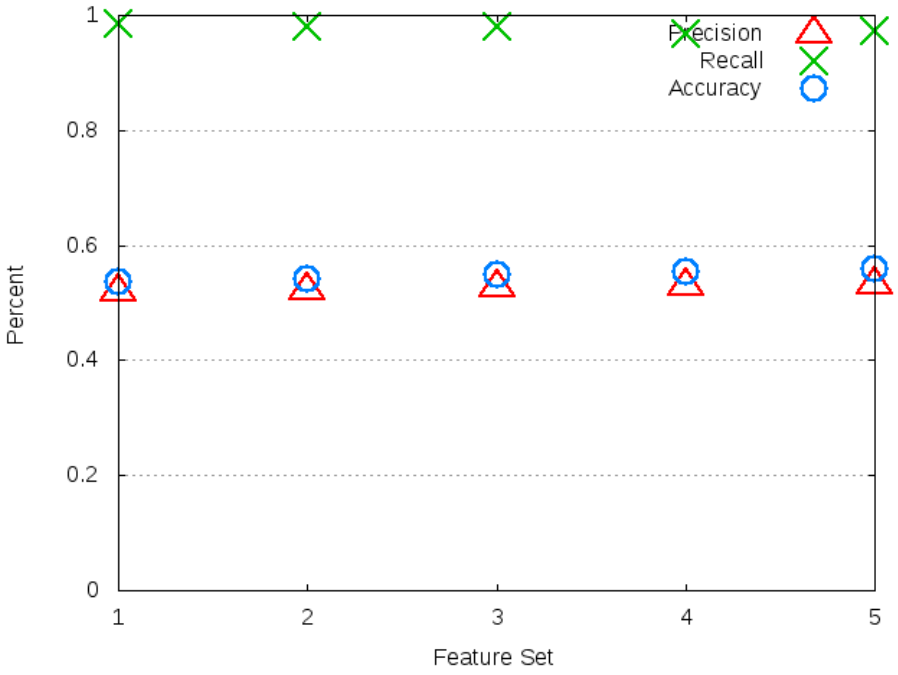


Figure A.94: Feature for arquillian-core using RF

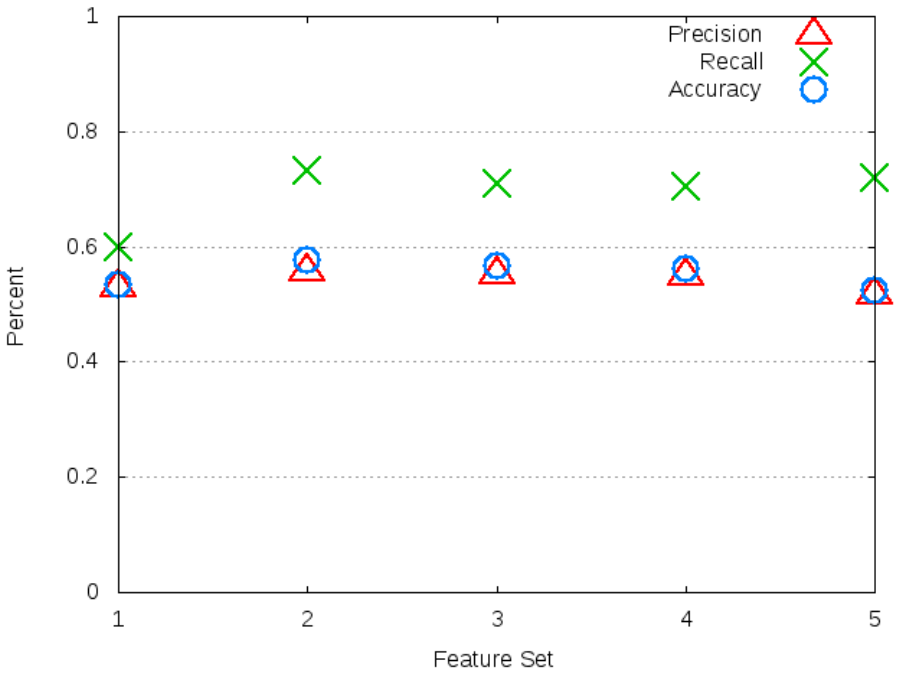


Figure A.95: Feature for blockly-android using RF

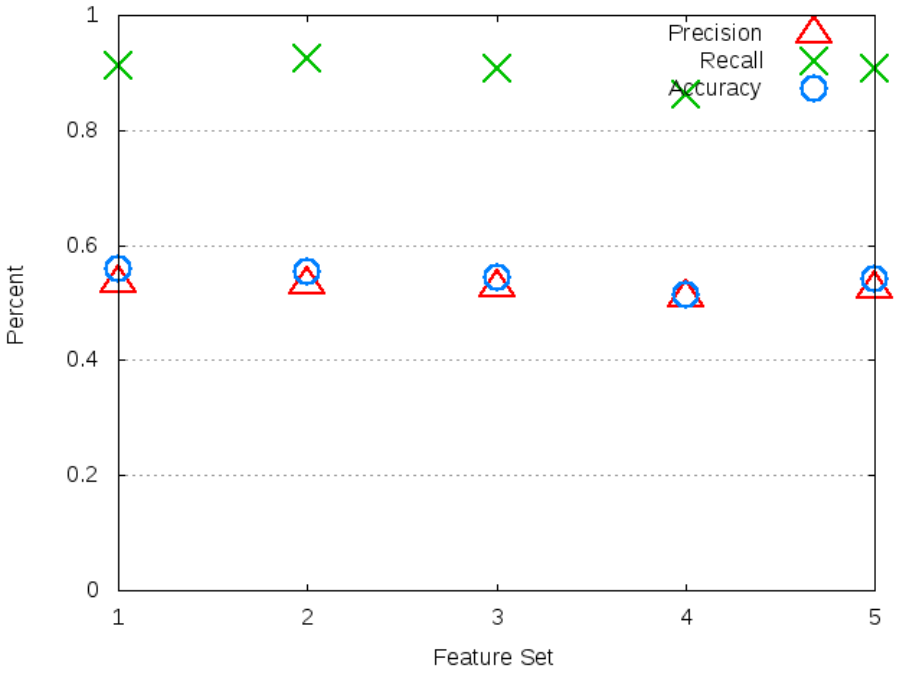


Figure A.96: Feature for brave using RF

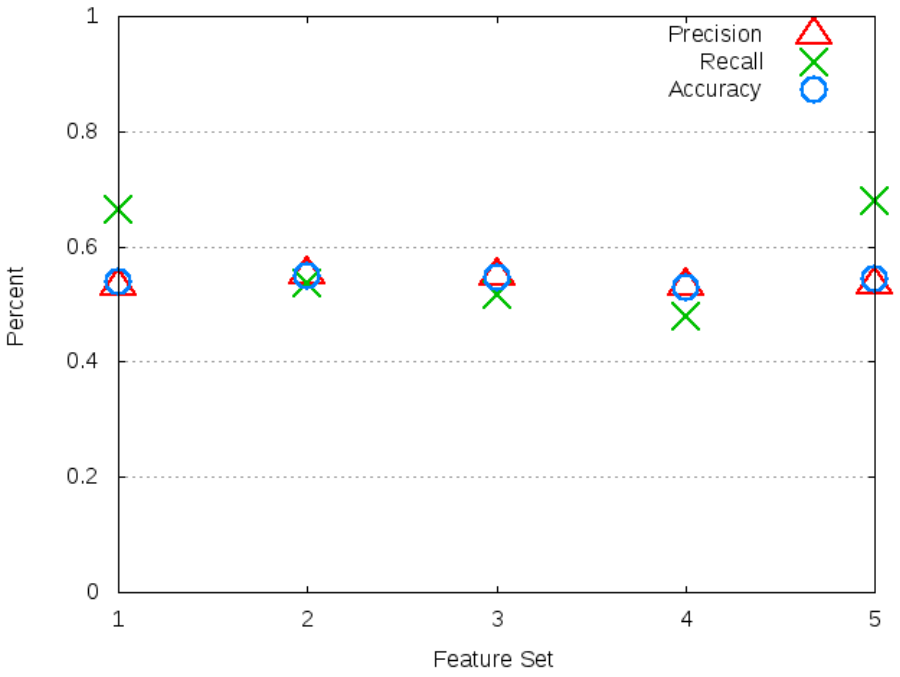


Figure A.97: Feature for cardslib using RF

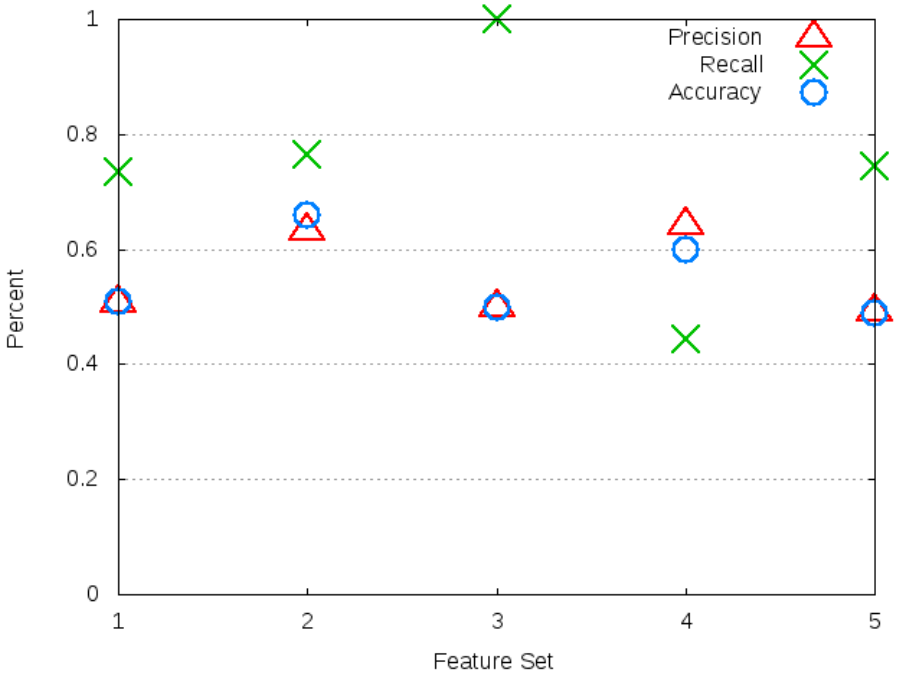


Figure A.98: Feature for dagger using RF

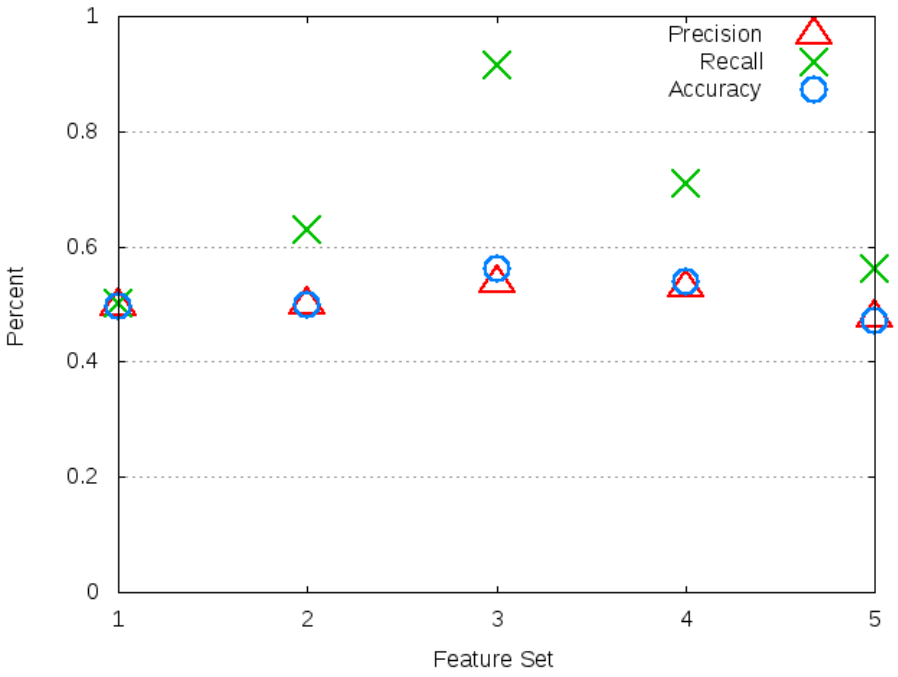


Figure A.99: Feature for deeplearning4j using RF

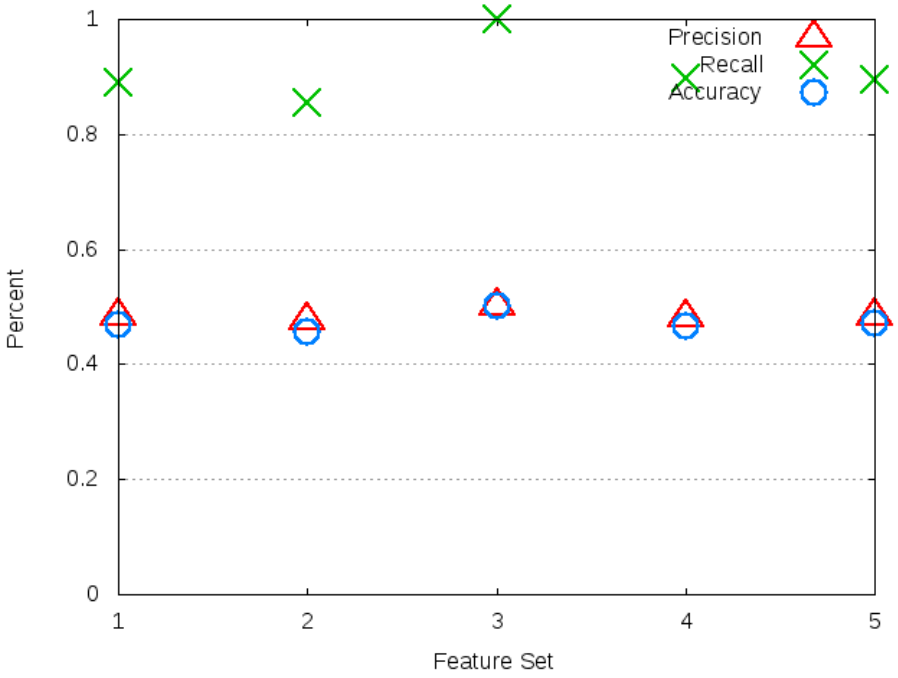


Figure A.100: Feature for fresco using RF

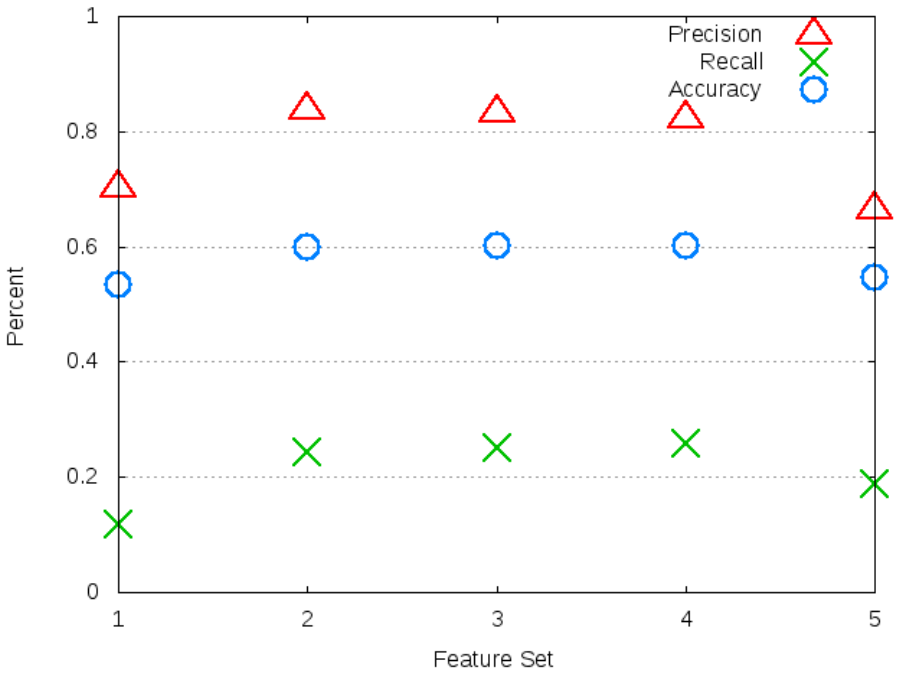


Figure A.101: Feature for governator using RF

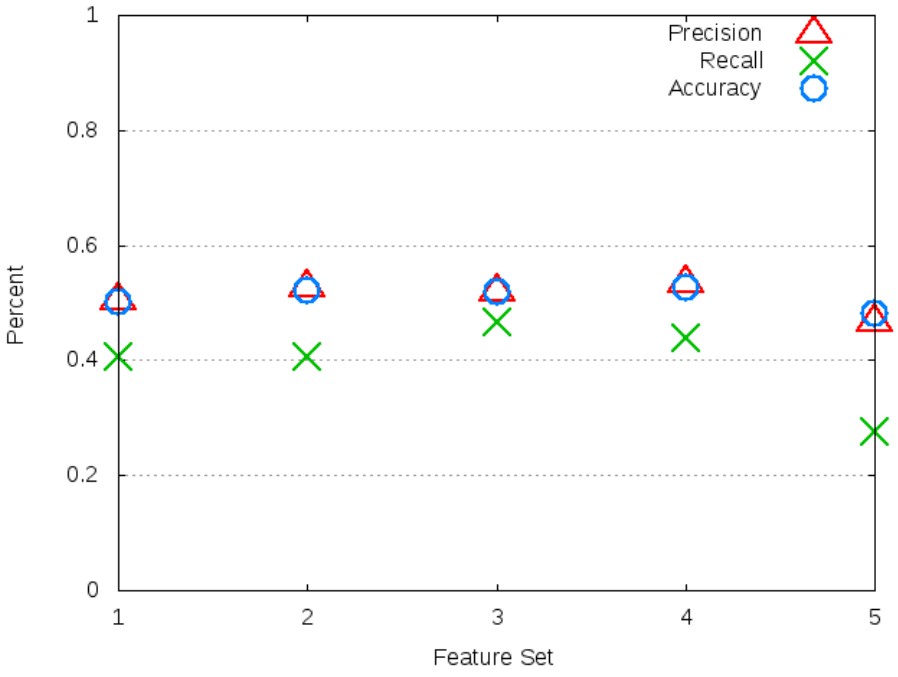


Figure A.102: Feature for greenDAO using RF

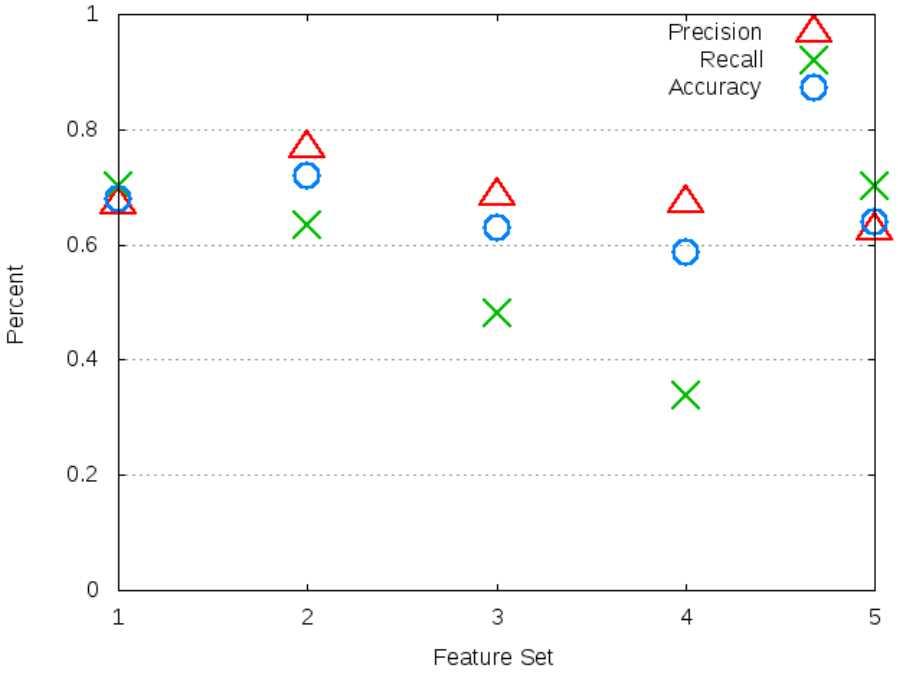


Figure A.103: Feature for http-request using RF

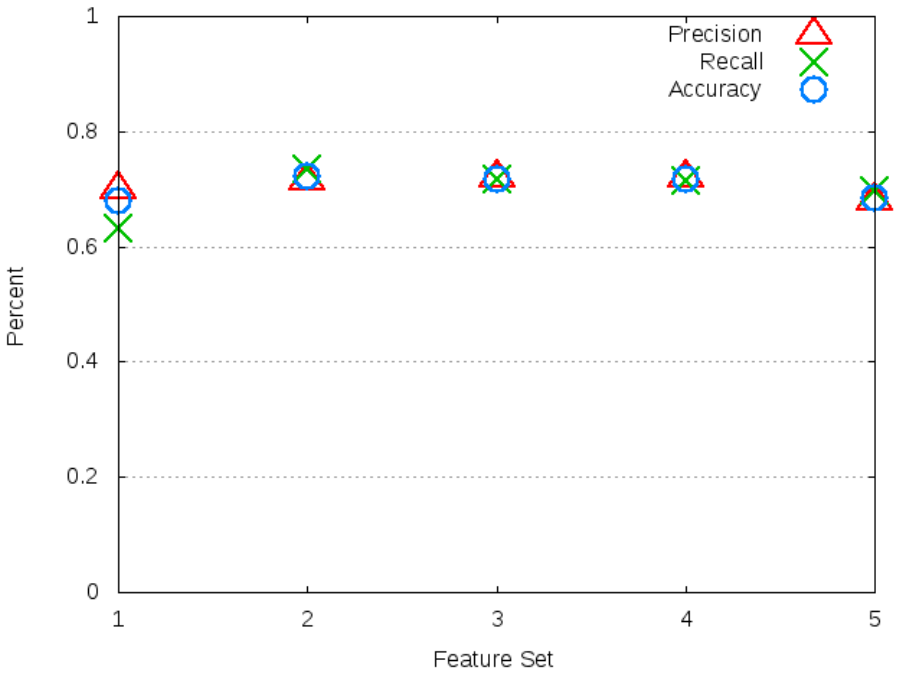


Figure A.104: Feature for ion using RF

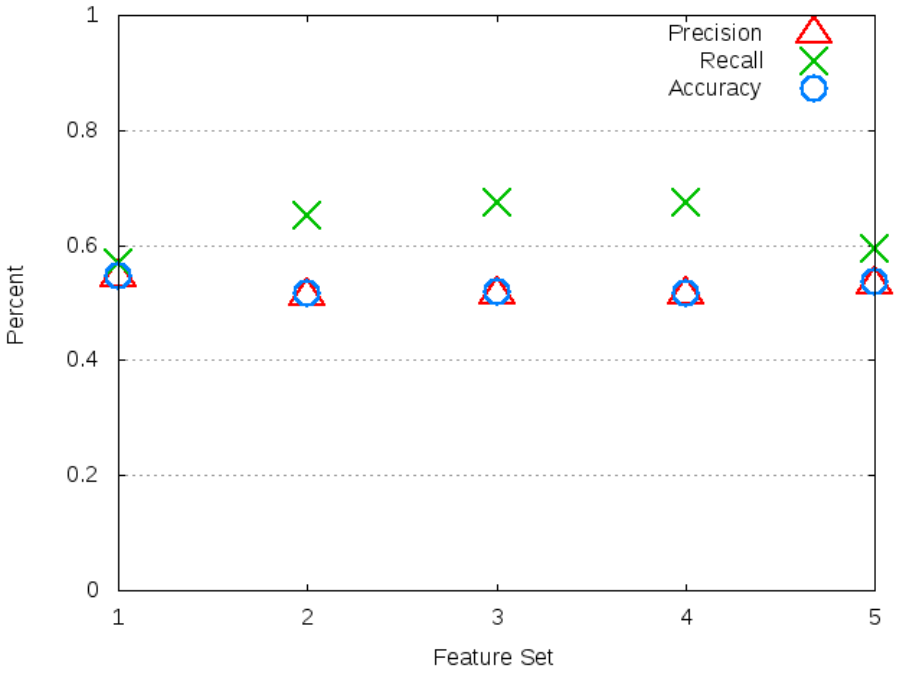


Figure A.105: Feature for jadx using RF

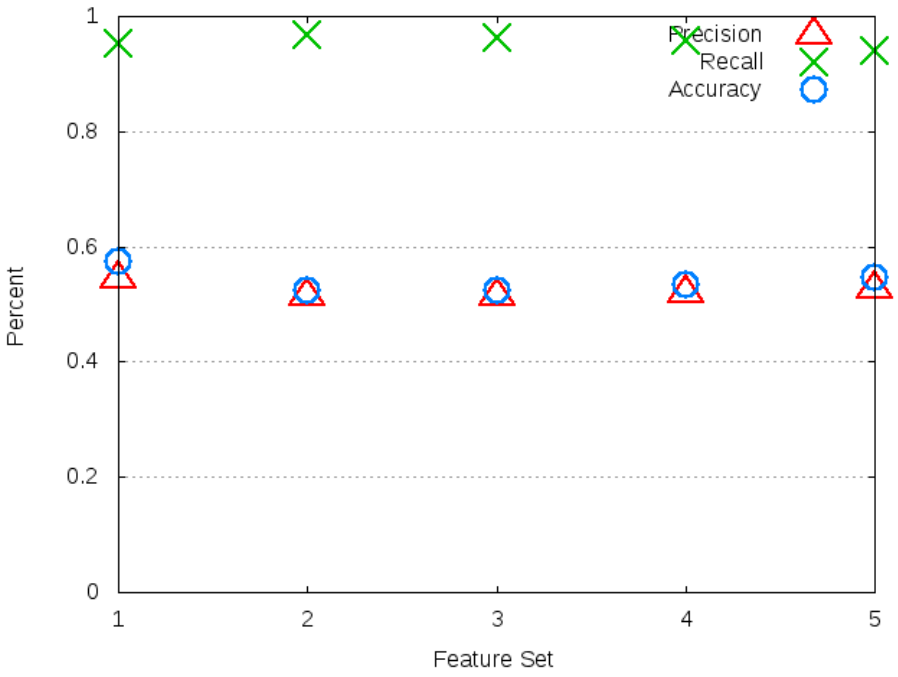


Figure A.106: Feature for mapstruct using RF

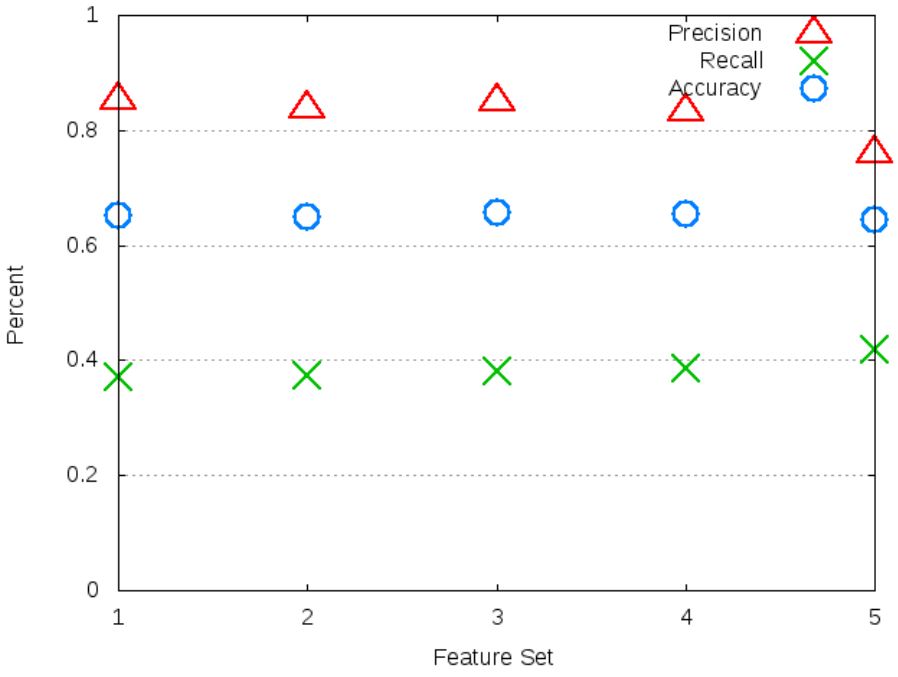


Figure A.107: Feature for nettosphere using RF

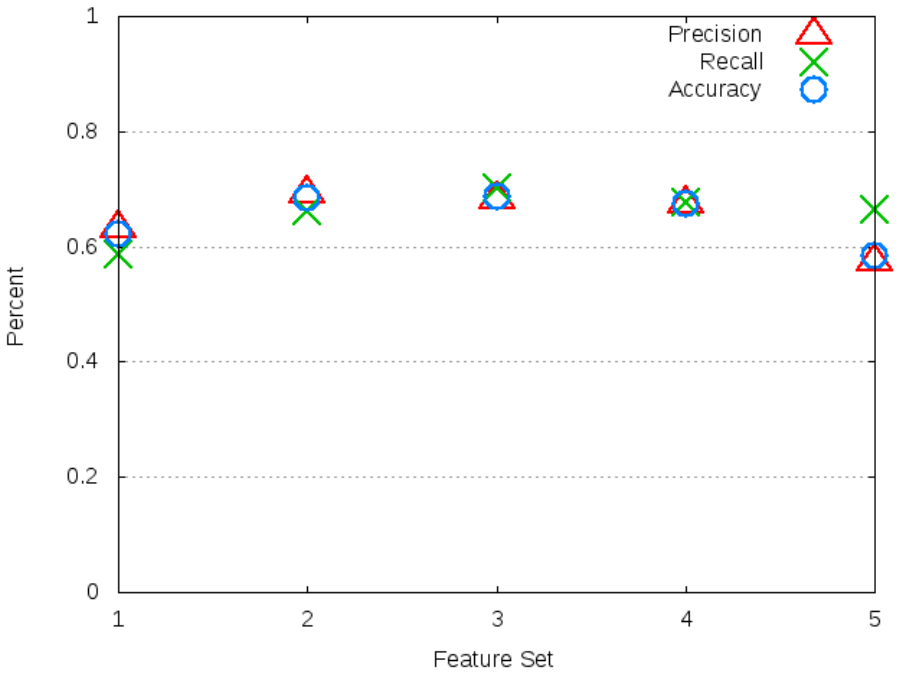


Figure A.108: Feature for parceler using RF

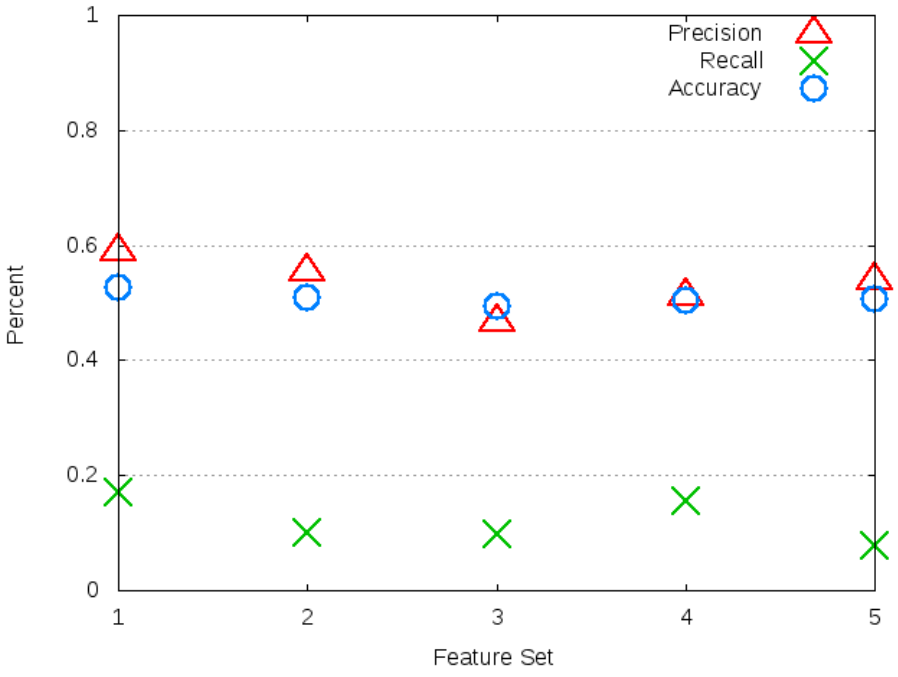


Figure A.109: Feature for retrolambda using RF

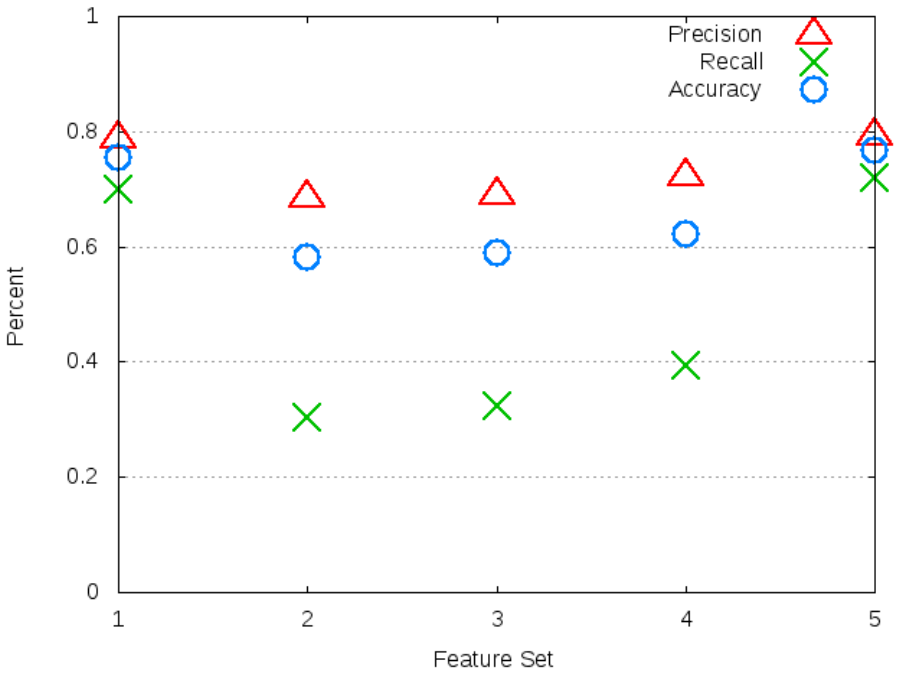


Figure A.110: Feature for ShowcaseView using RF

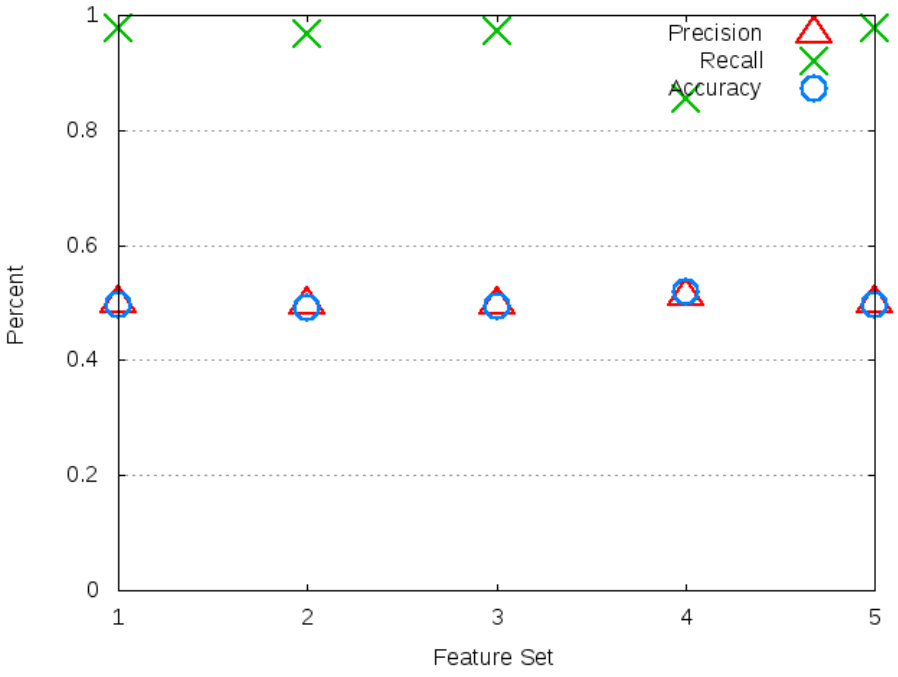


Figure A.111: Feature for smile using RF

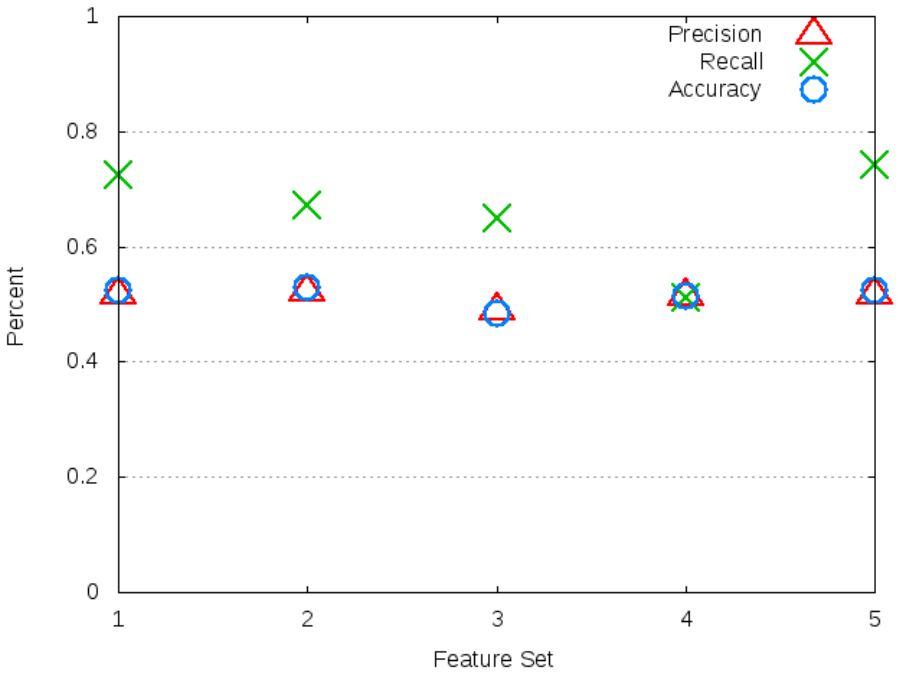


Figure A.112: Feature for spark using RF

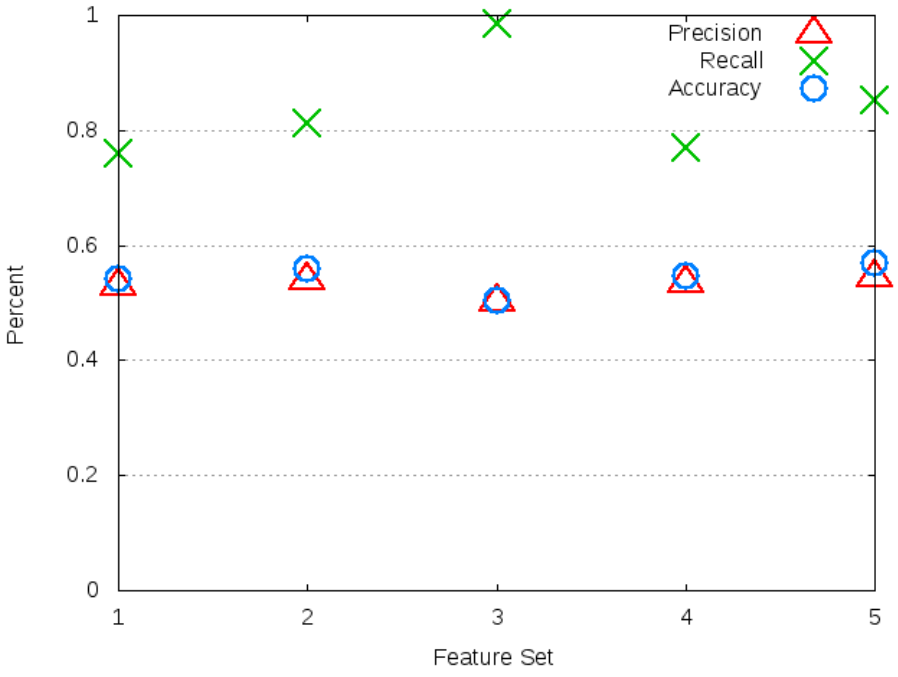


Figure A.113: Feature for storm using RF

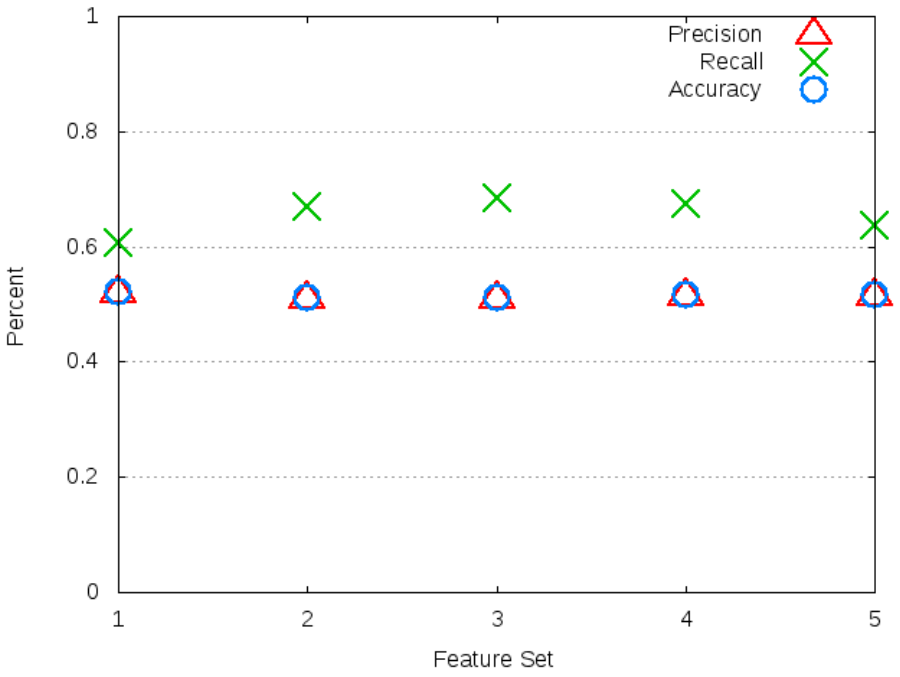


Figure A.114: Feature for tempto using RF

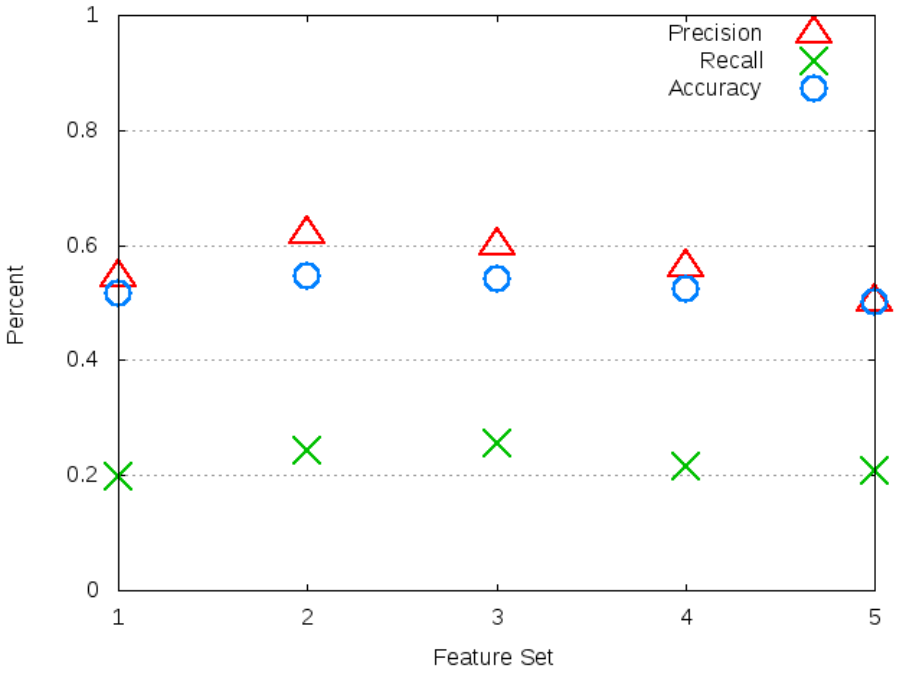


Figure A.115: Feature for yardstick using RF

A.3 Experiment 3

A.3.1 Support Vector Machine

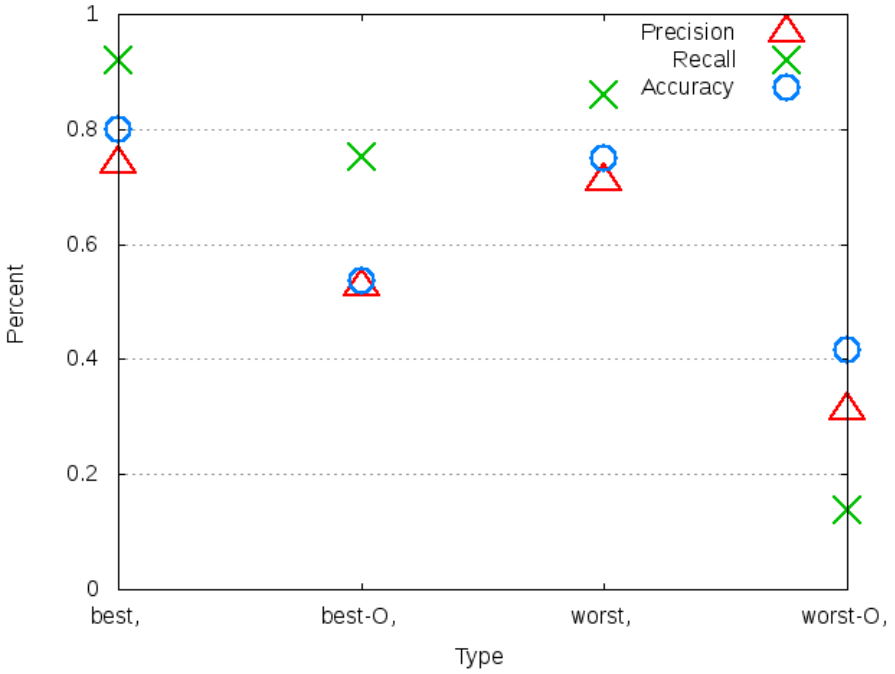


Figure A.116: Oversampling for acra using SVM

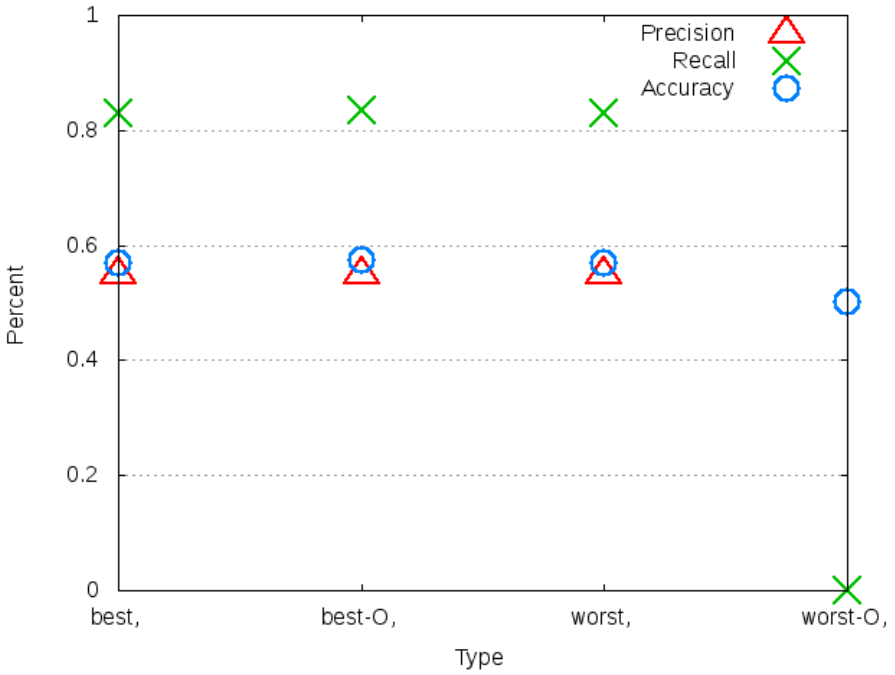


Figure A.117: Oversampling for arquillian-core using SVM

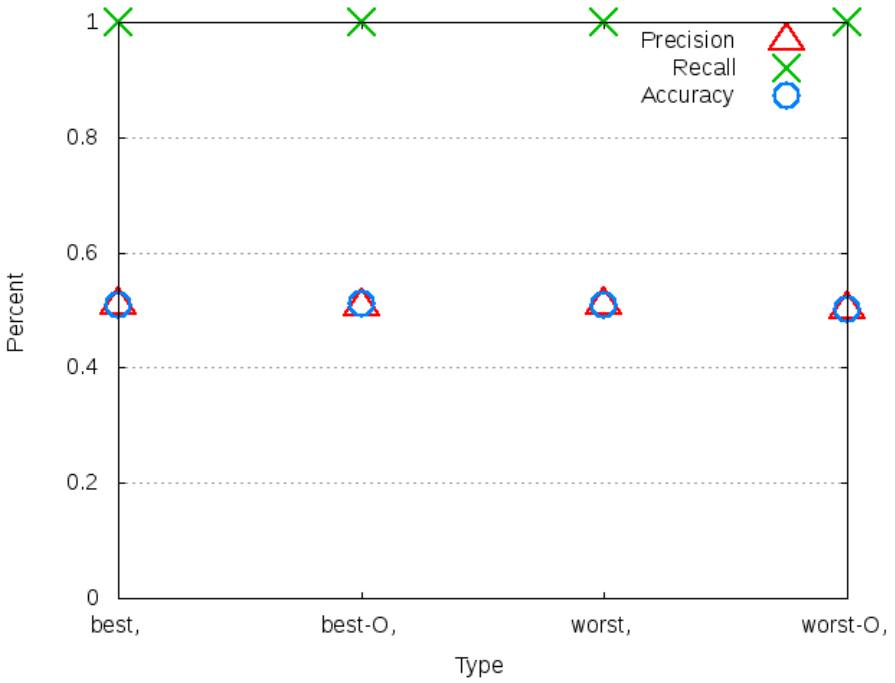


Figure A.118: Oversampling for blockly-android using SVM

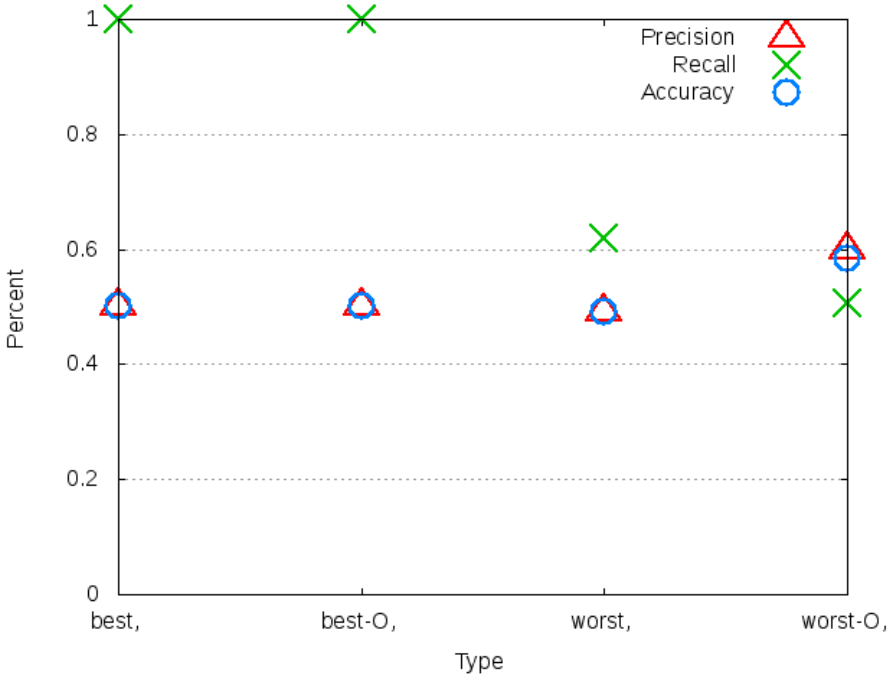


Figure A.119: Oversampling for brave using SVM

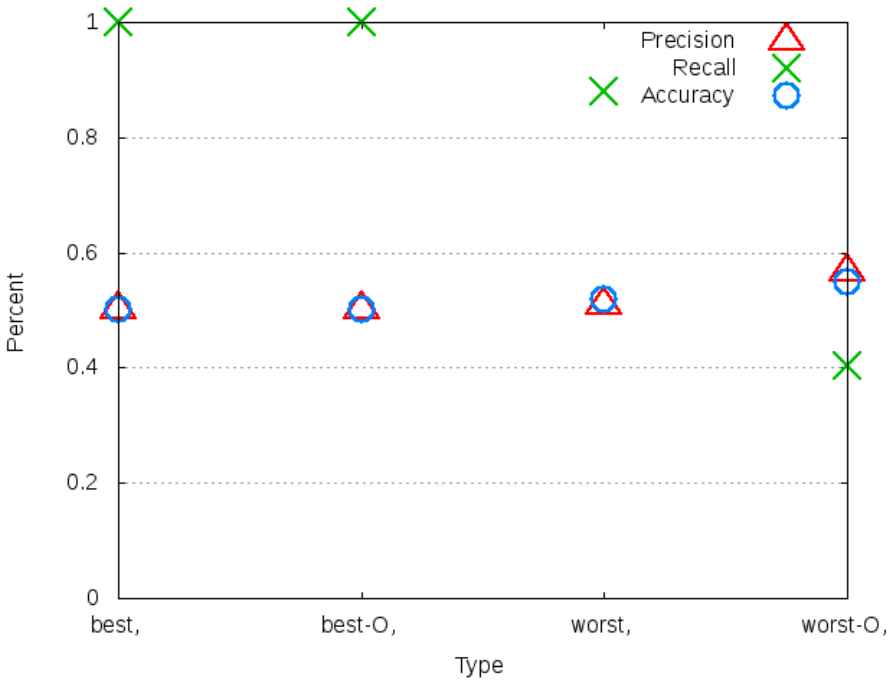


Figure A.120: Oversampling for cardslib using SVM

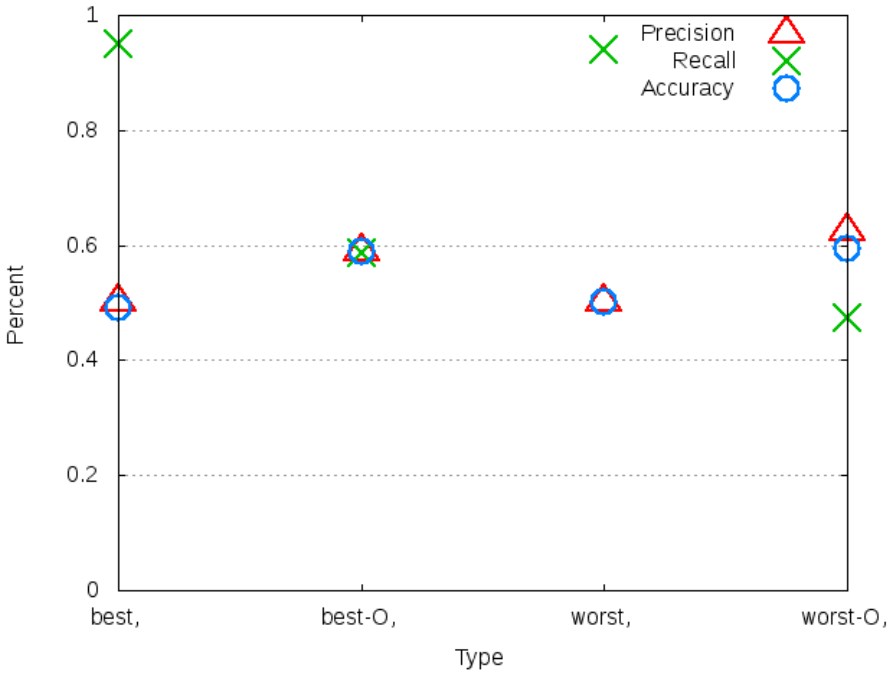


Figure A.121: Oversampling for dagger using SVM

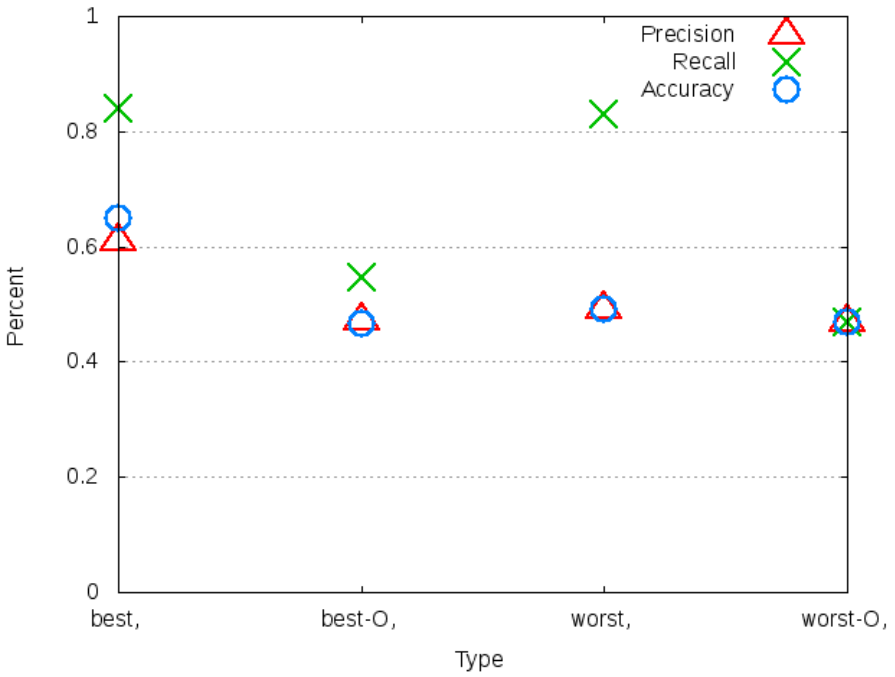


Figure A.122: Oversampling for deeplearning4j using SVM

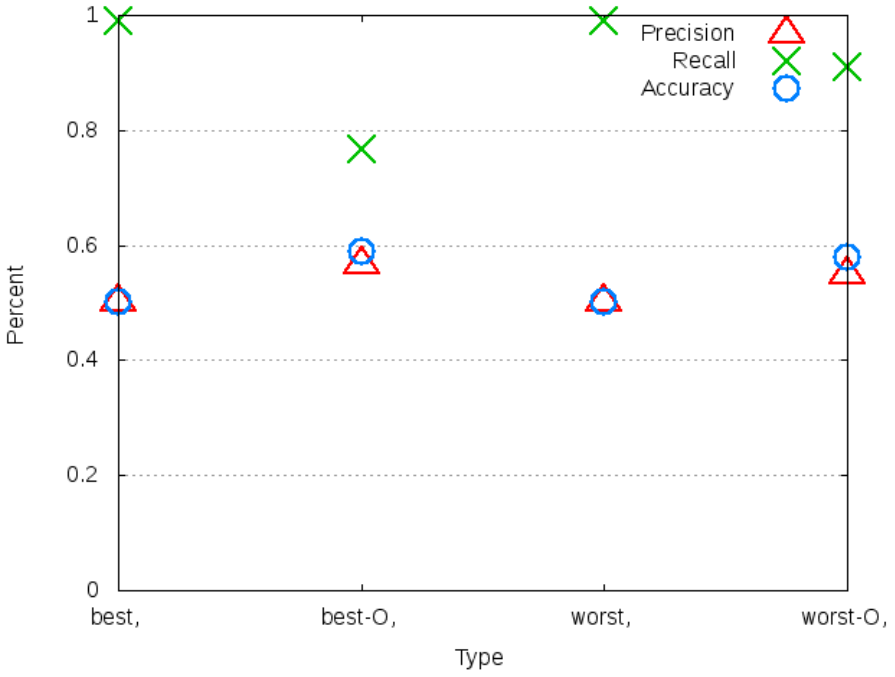


Figure A.123: Oversampling for fresco using SVM

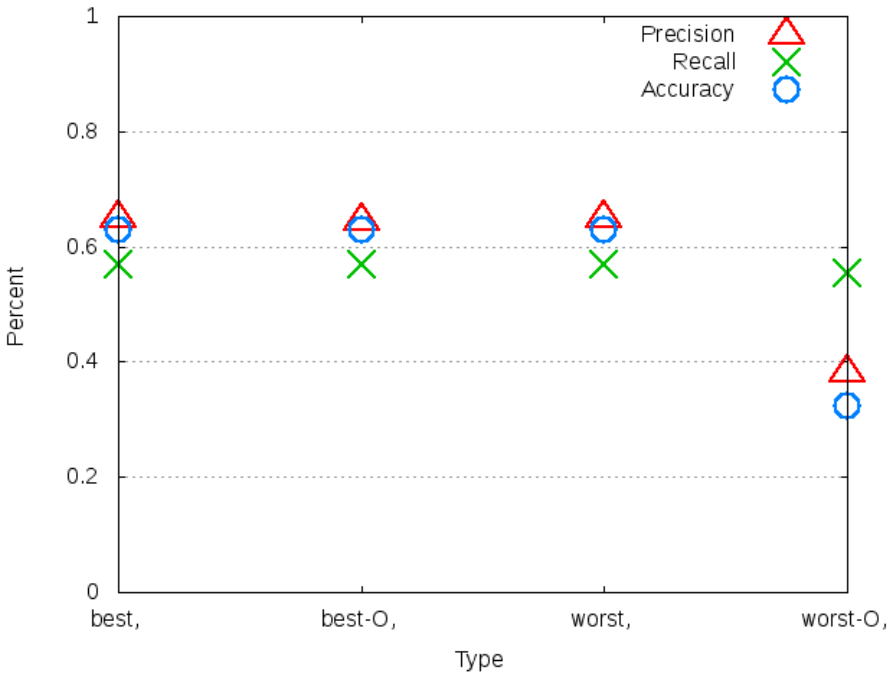


Figure A.124: Oversampling for governator using SVM

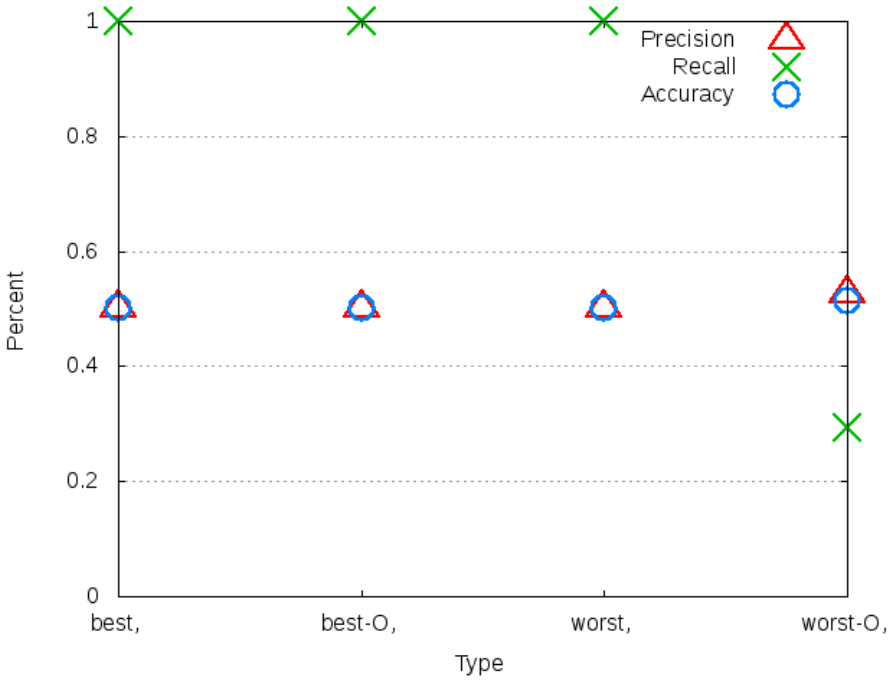


Figure A.125: Oversampling for greenDAO using SVM

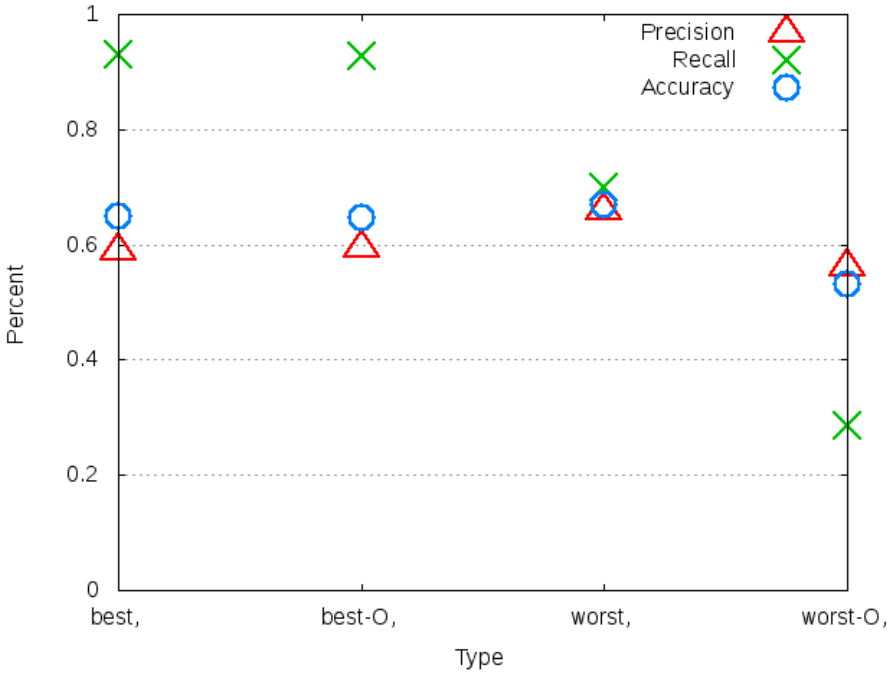


Figure A.126: Oversampling for http-request using SVM

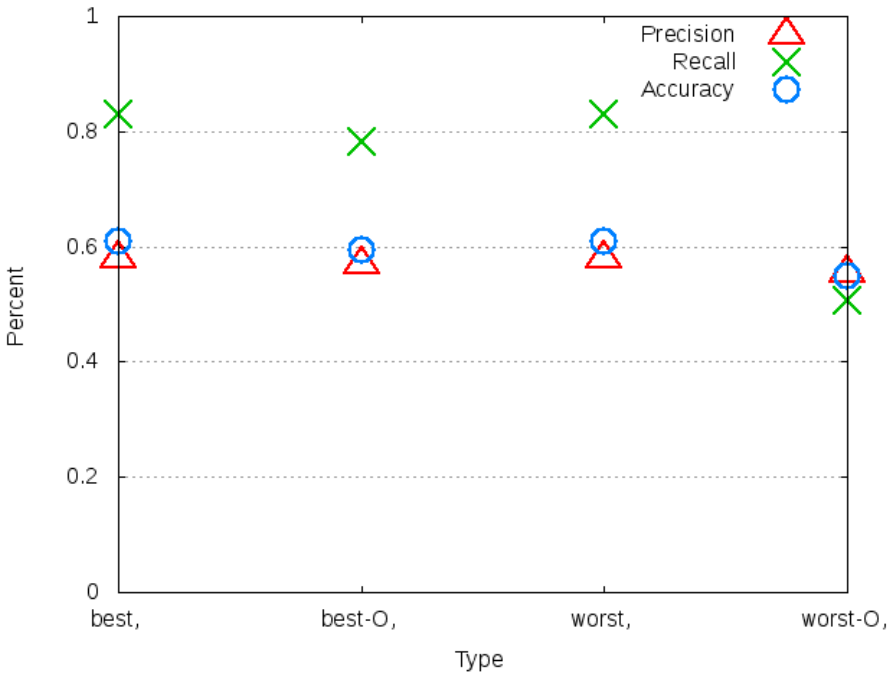


Figure A.127: Oversampling for ion using SVM

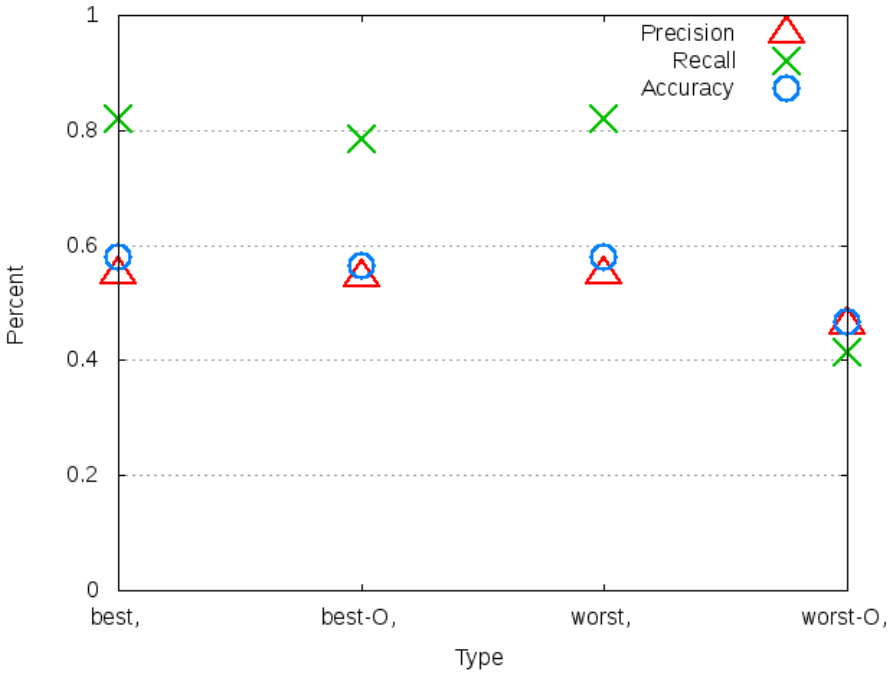


Figure A.128: Oversampling for jadx using SVM

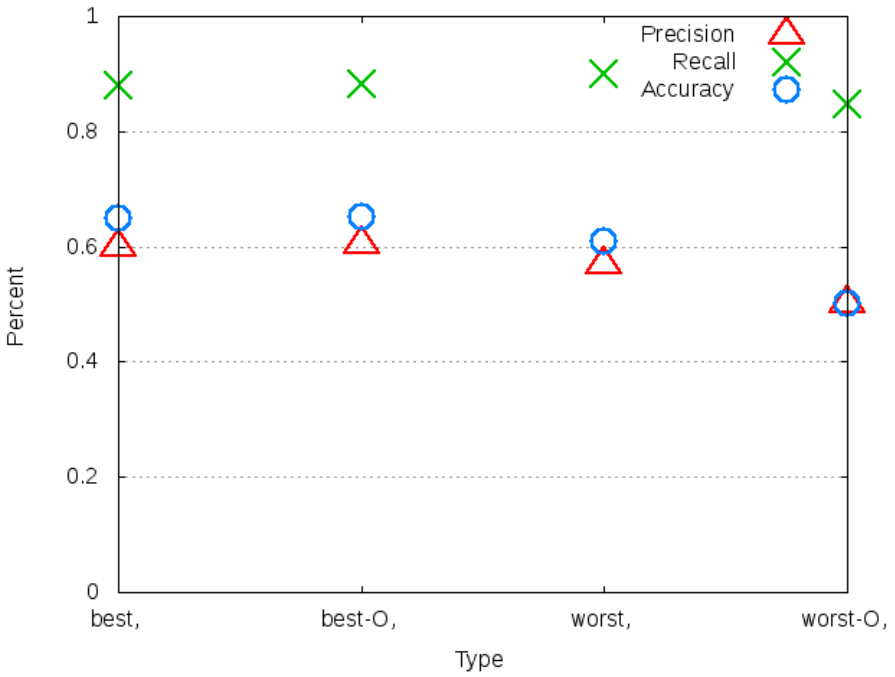


Figure A.129: Oversampling for mapstruct using SVM

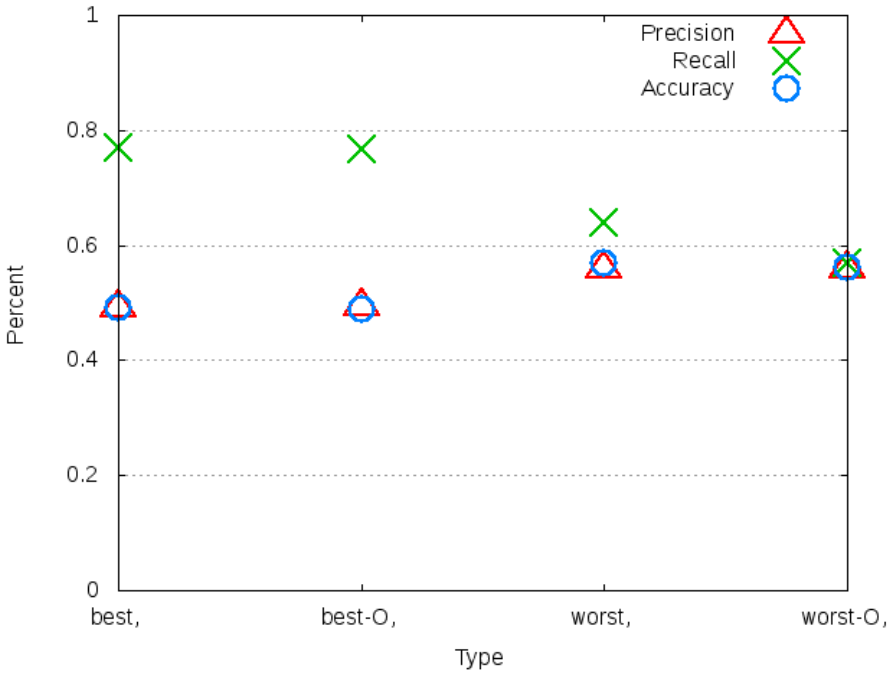


Figure A.130: Oversampling for nettosphere using SVM

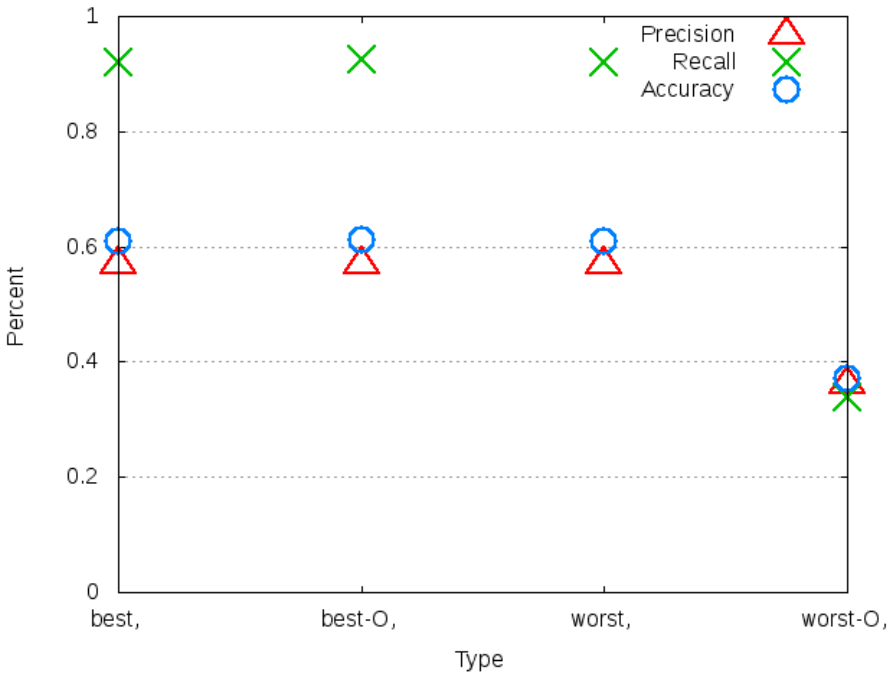


Figure A.131: Oversampling for parceler using SVM

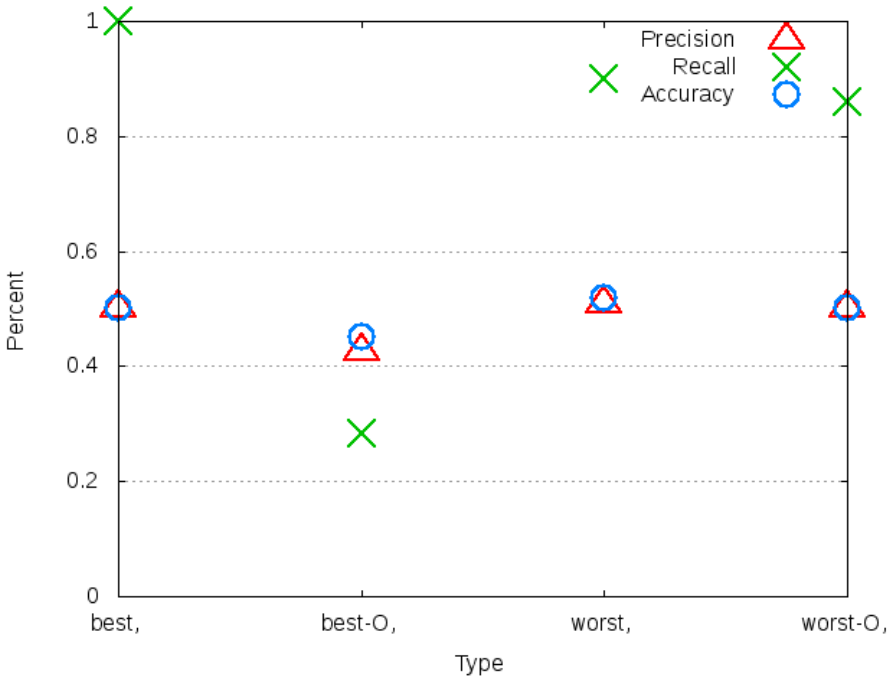


Figure A.132: Oversampling for retrolambda using SVM

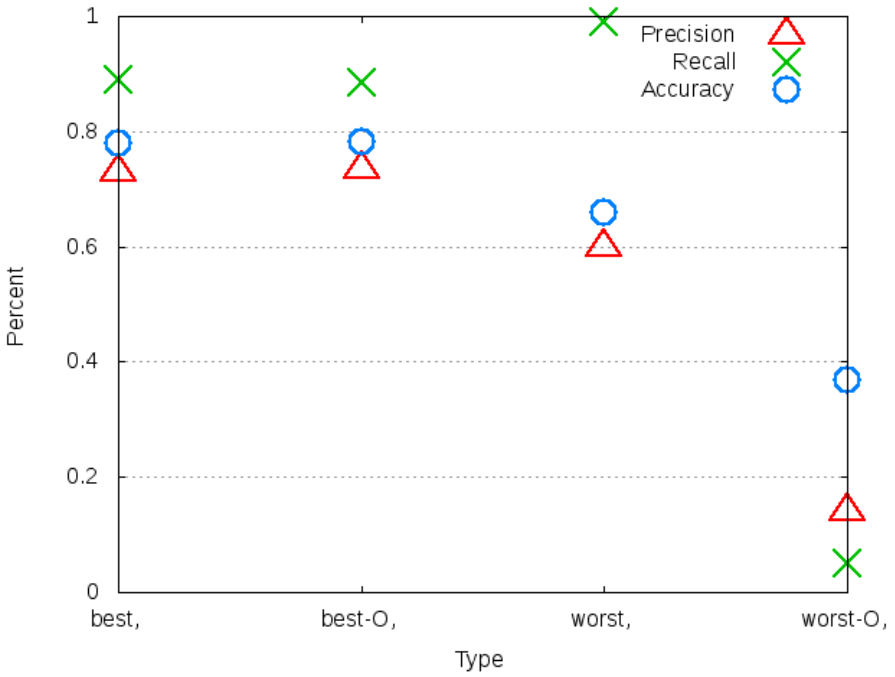


Figure A.133: Oversampling for ShowcaseView using SVM

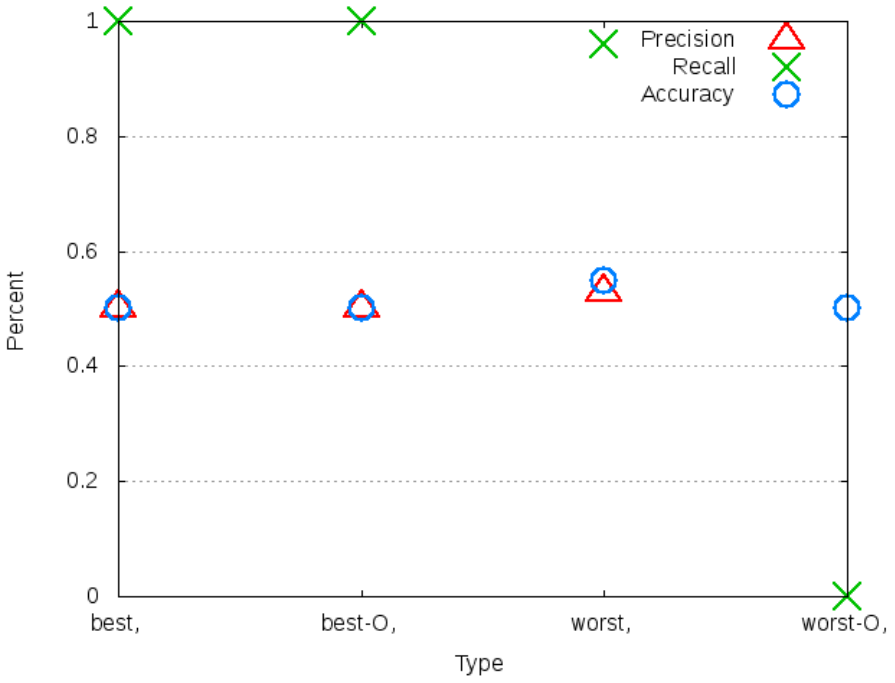


Figure A.134: Oversampling for smile using SVM

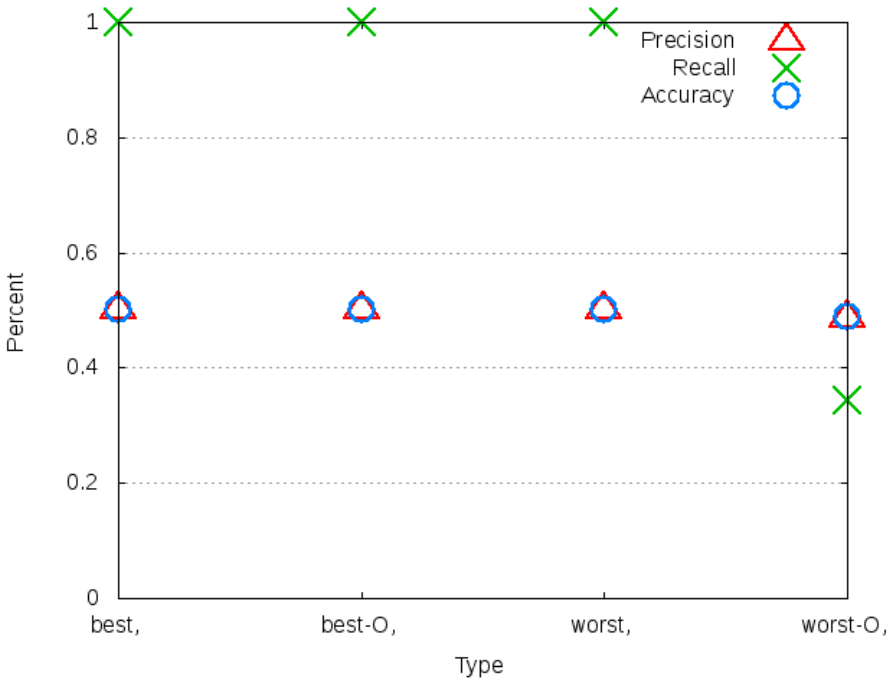


Figure A.135: Oversampling for spark using SVM

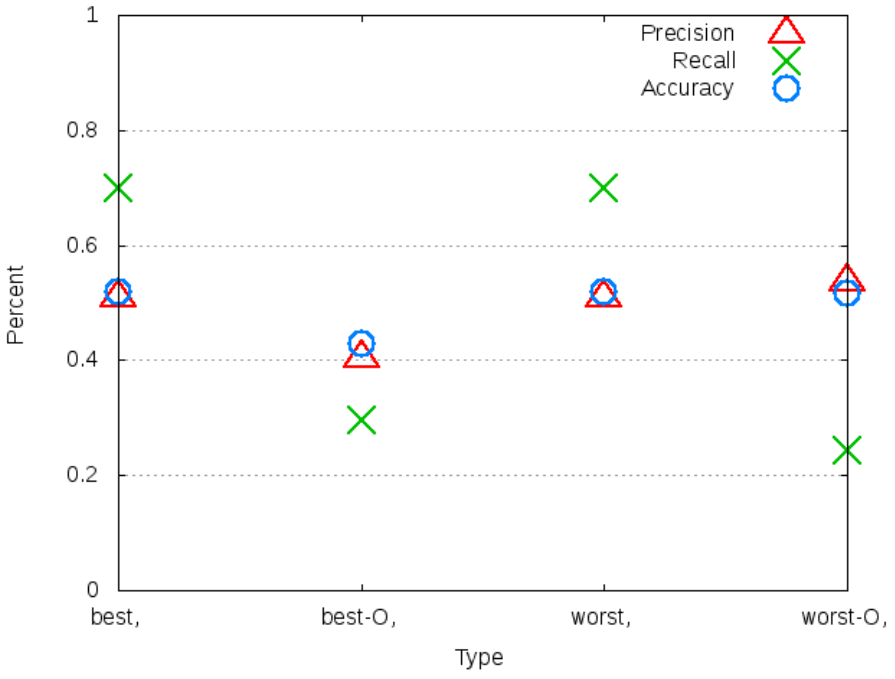


Figure A.136: Oversampling for storm using SVM

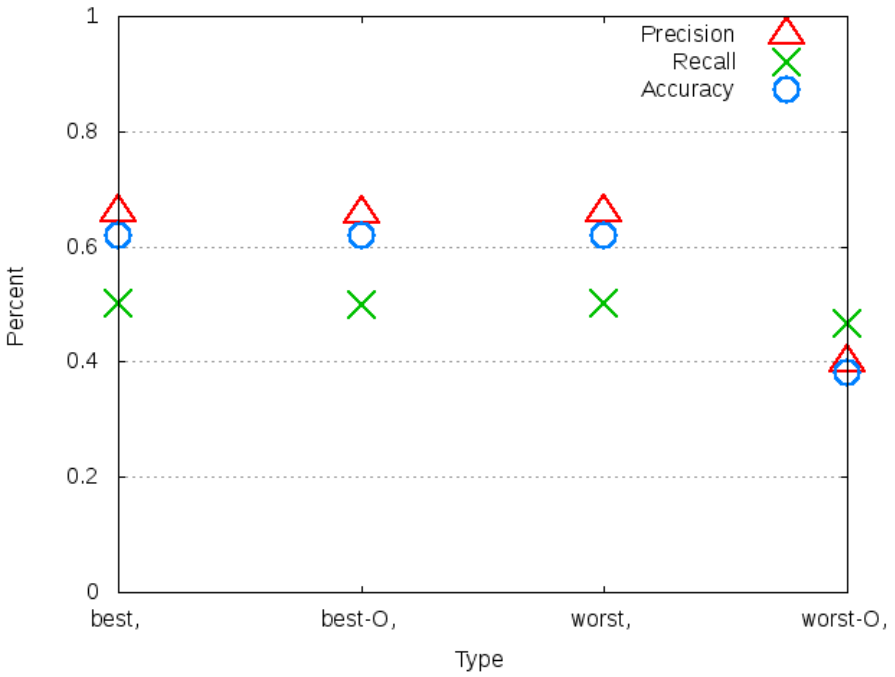


Figure A.137: Oversampling for tempto using SVM

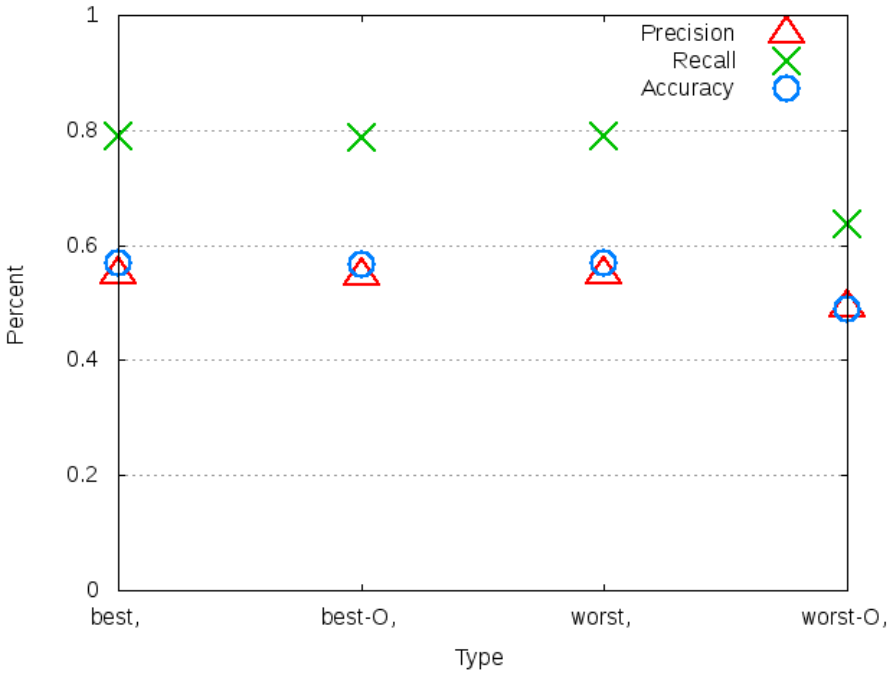


Figure A.138: Oversampling for yardstick using SVM

A.3.2 Random Forest

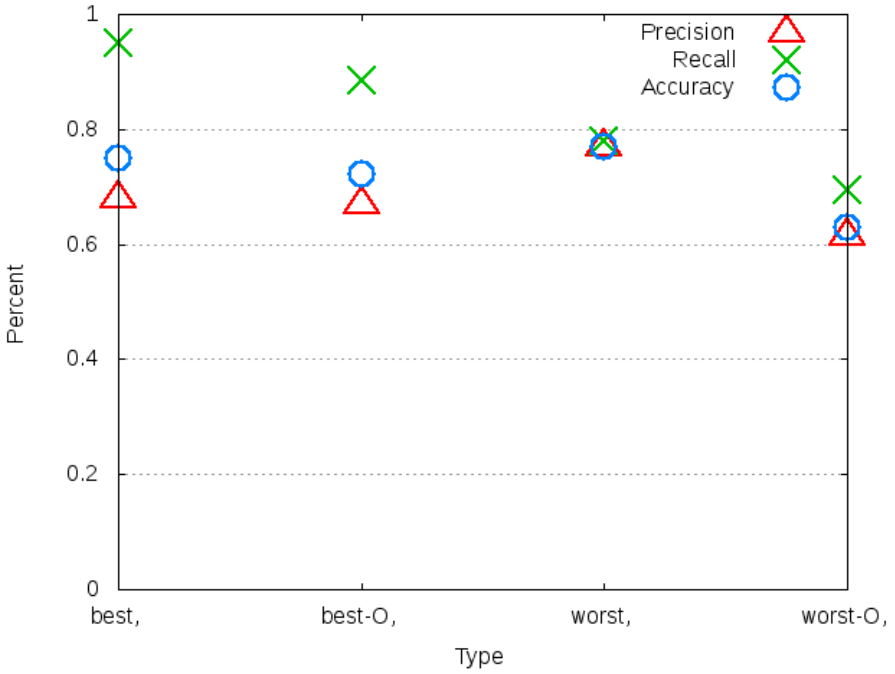


Figure A.139: Oversampling for acra using RF

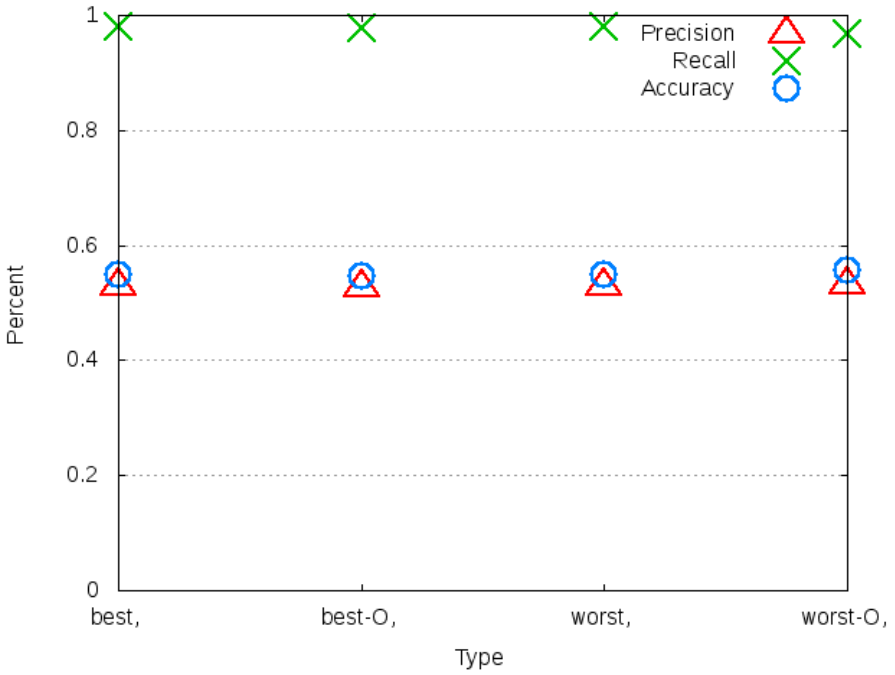


Figure A.140: Oversampling for arquillian-core using RF

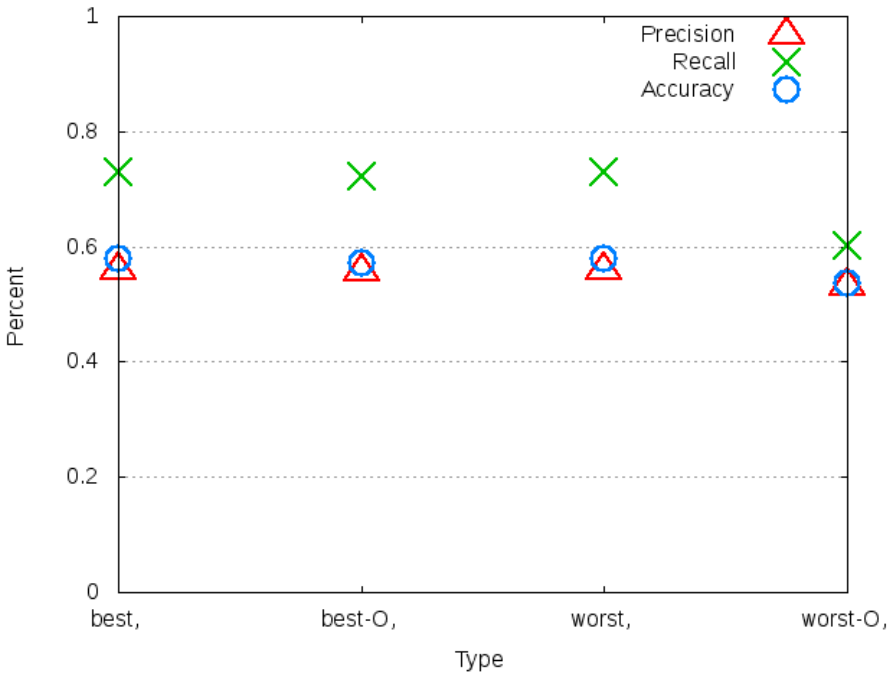


Figure A.141: Oversampling for blockly-android using RF

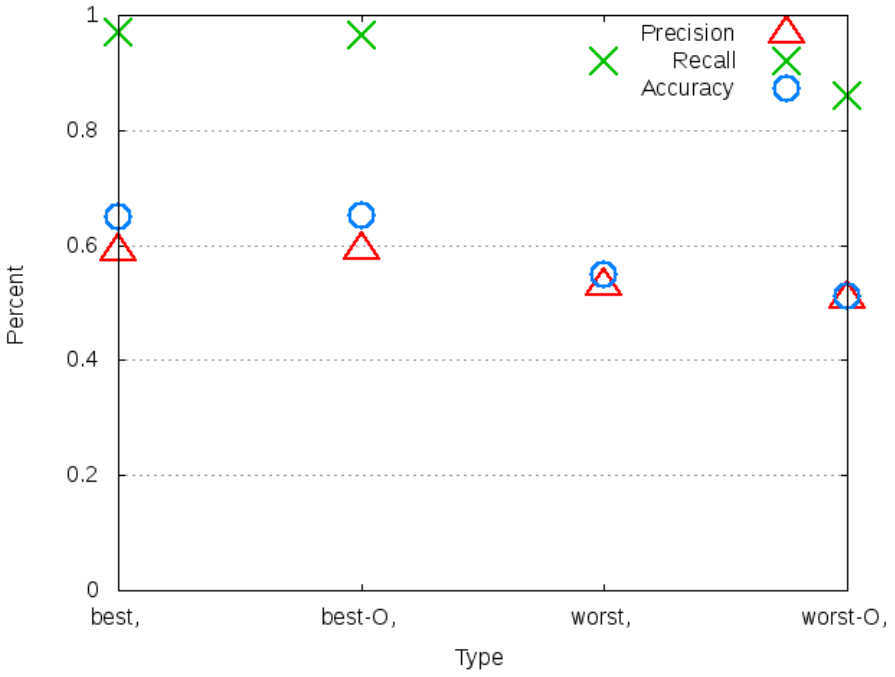


Figure A.142: Oversampling for brave using RF

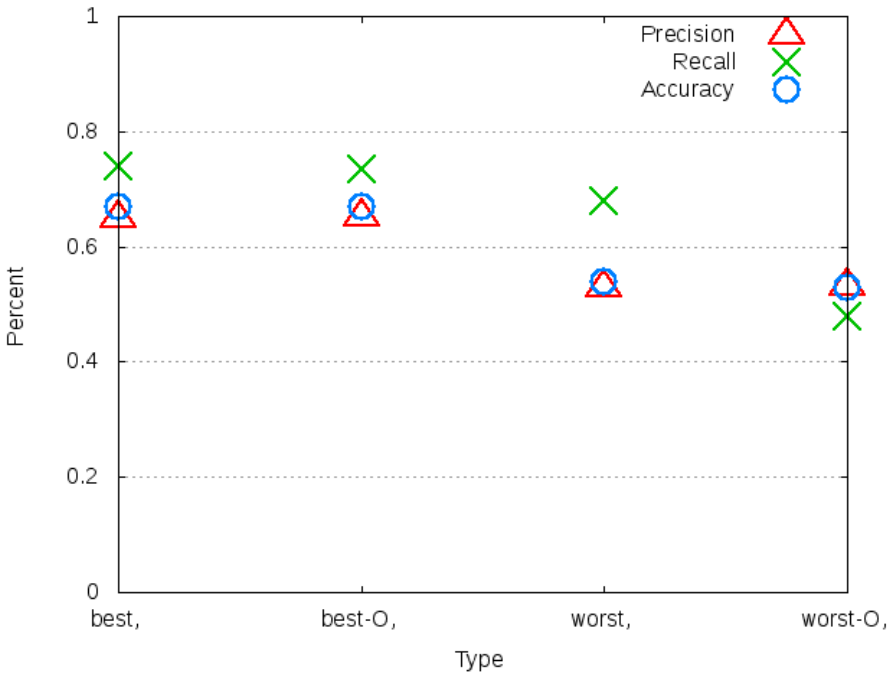


Figure A.143: Oversampling for cardslib using RF

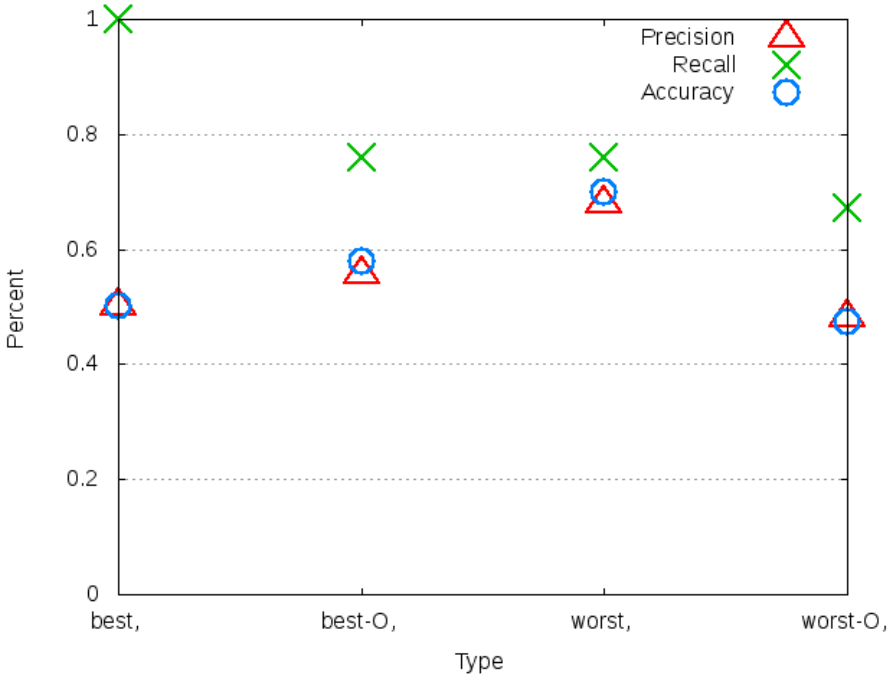


Figure A.144: Oversampling for dagger using RF

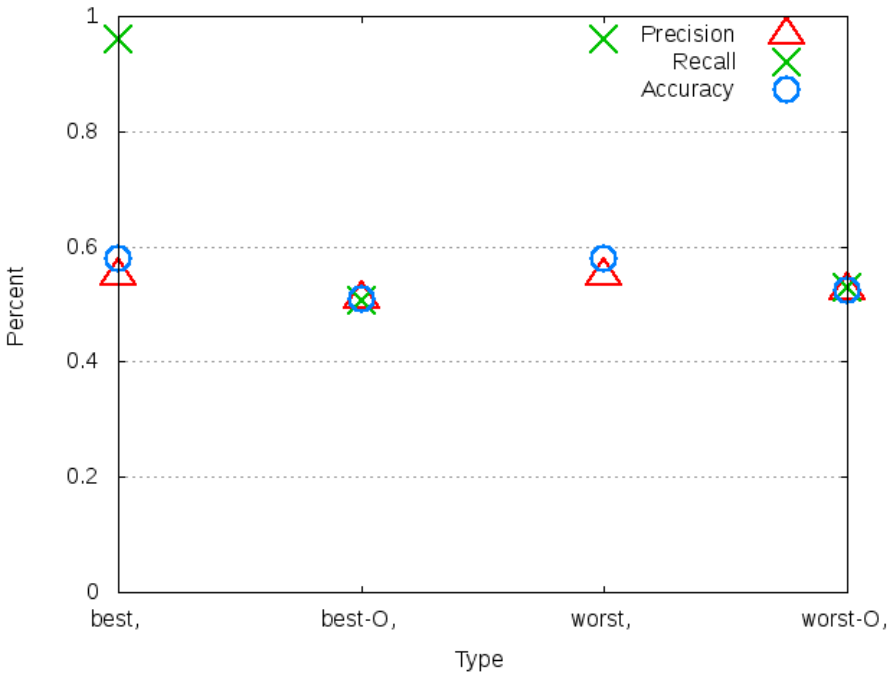


Figure A.145: Oversampling for deeplearning4j using RF

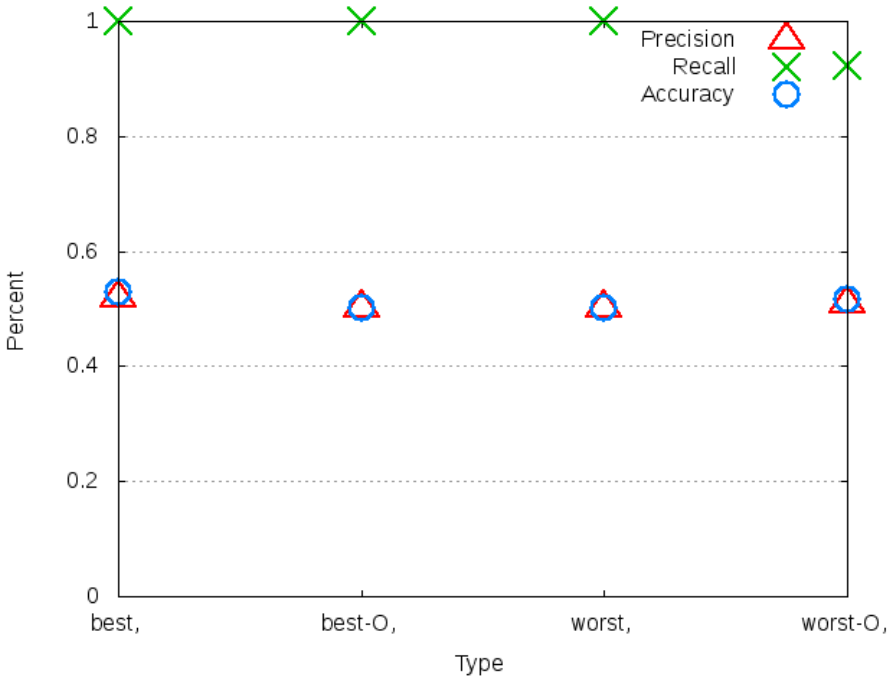


Figure A.146: Oversampling for fresco using RF

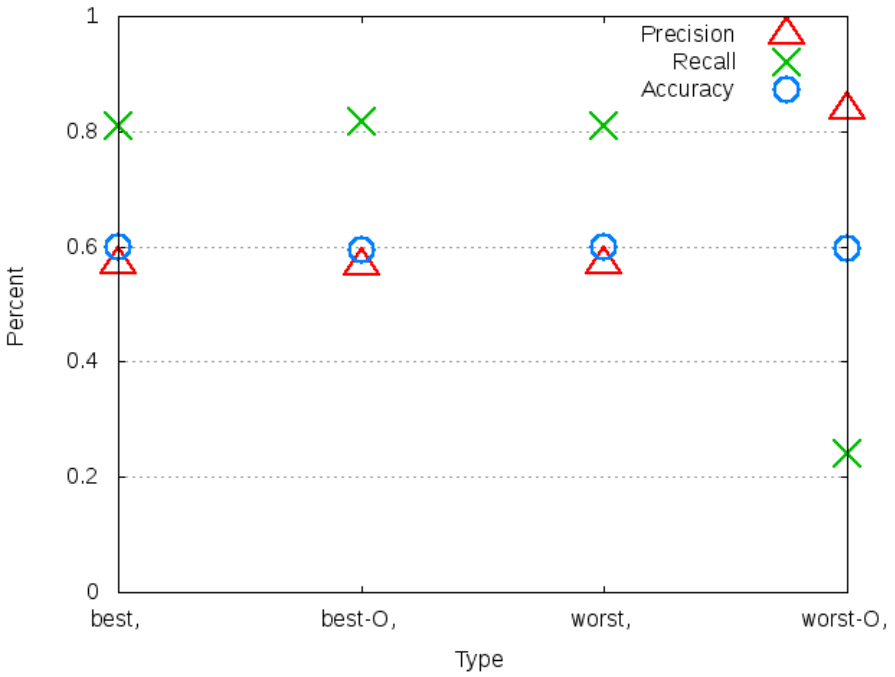


Figure A.147: Oversampling for governator using RF

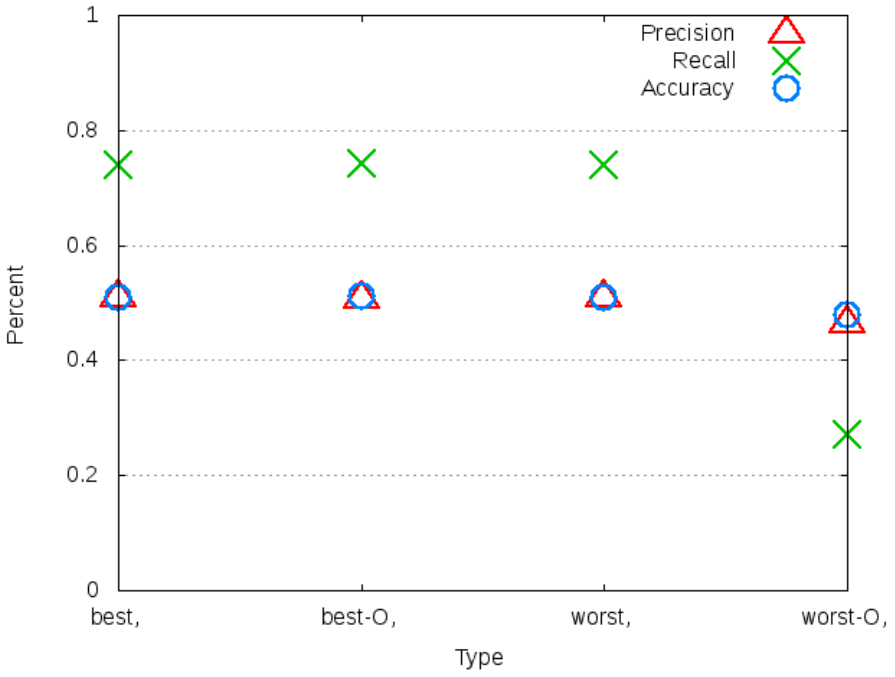


Figure A.148: Oversampling for greenDAO using RF

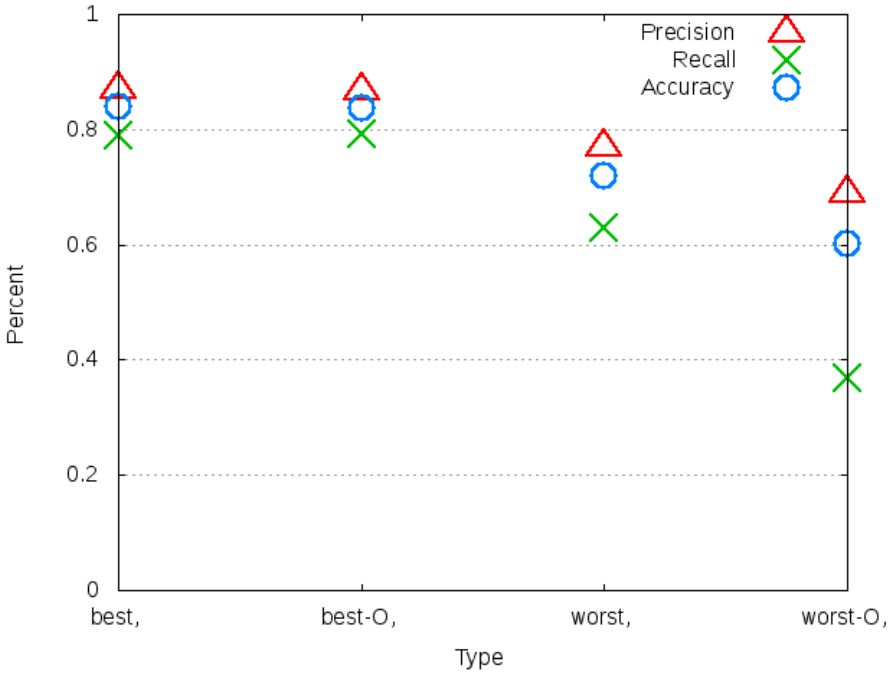


Figure A.149: Oversampling for http-request using RF

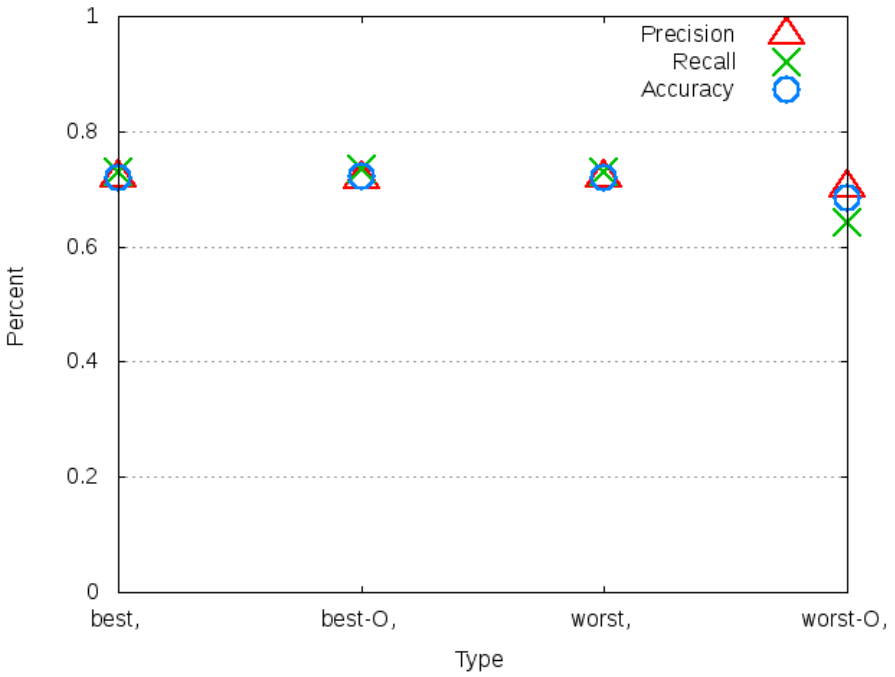


Figure A.150: Oversampling for ion using RF

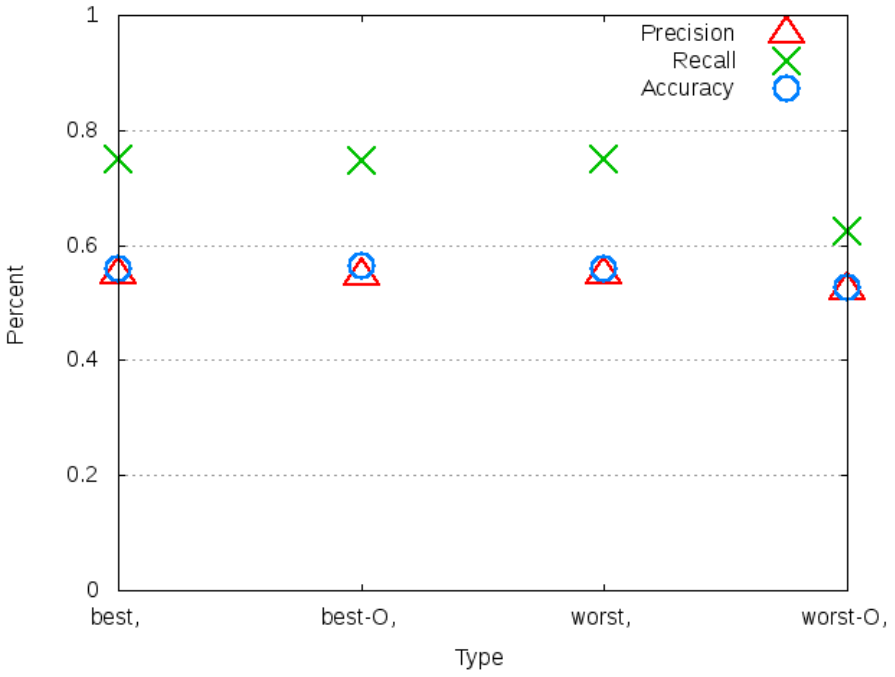


Figure A.151: Oversampling for jadx using RF

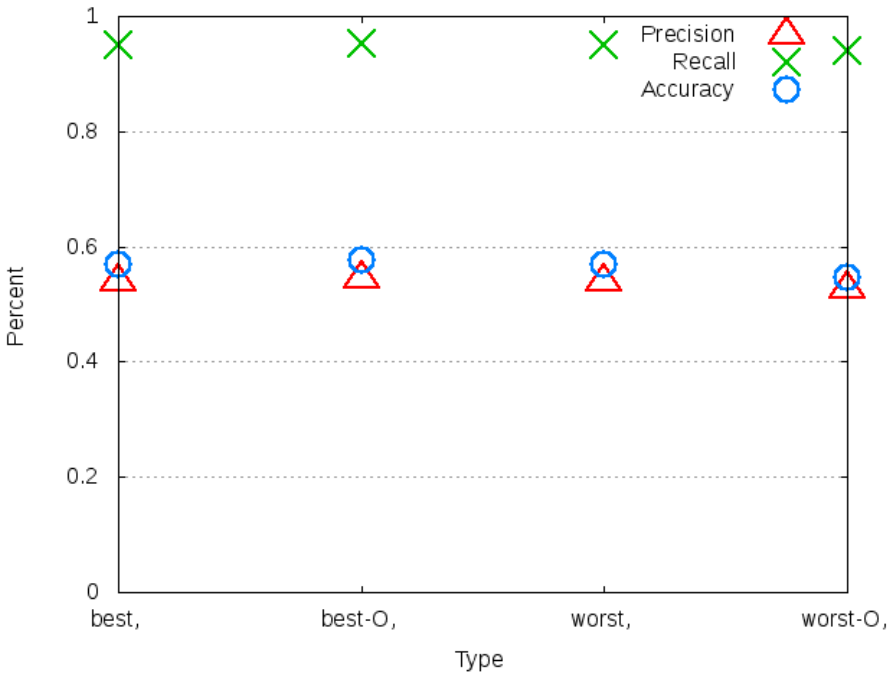


Figure A.152: Oversampling for mapstruct using RF

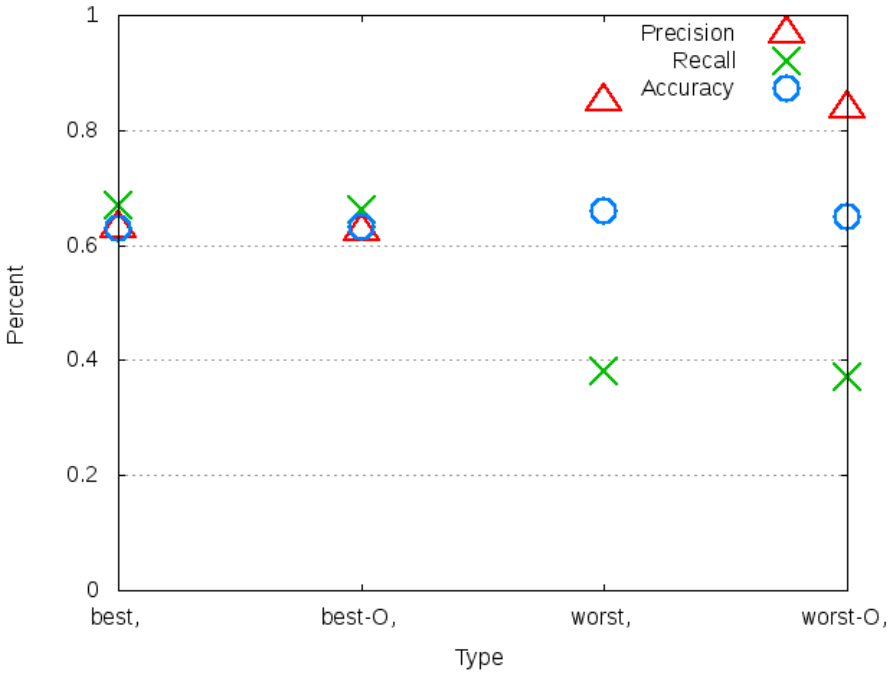


Figure A.153: Oversampling for nettosphere using RF

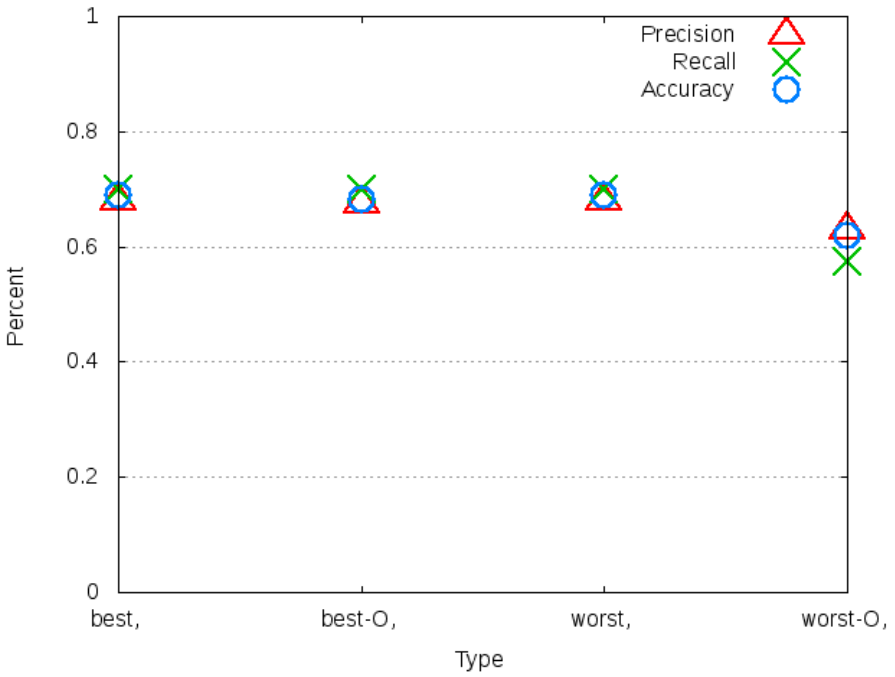


Figure A.154: Oversampling for parceler using RF

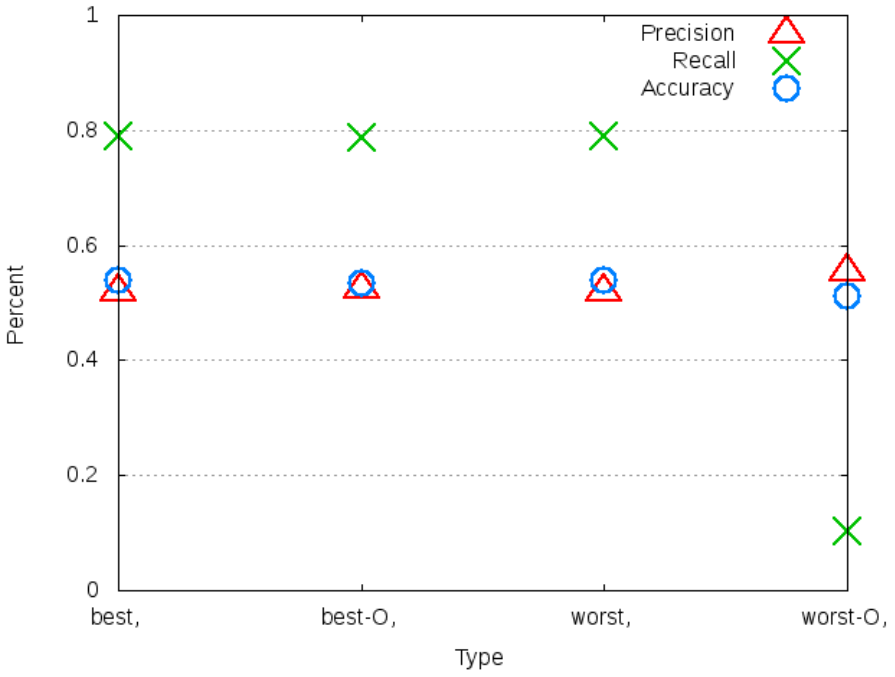


Figure A.155: Oversampling for retrolambda using RF

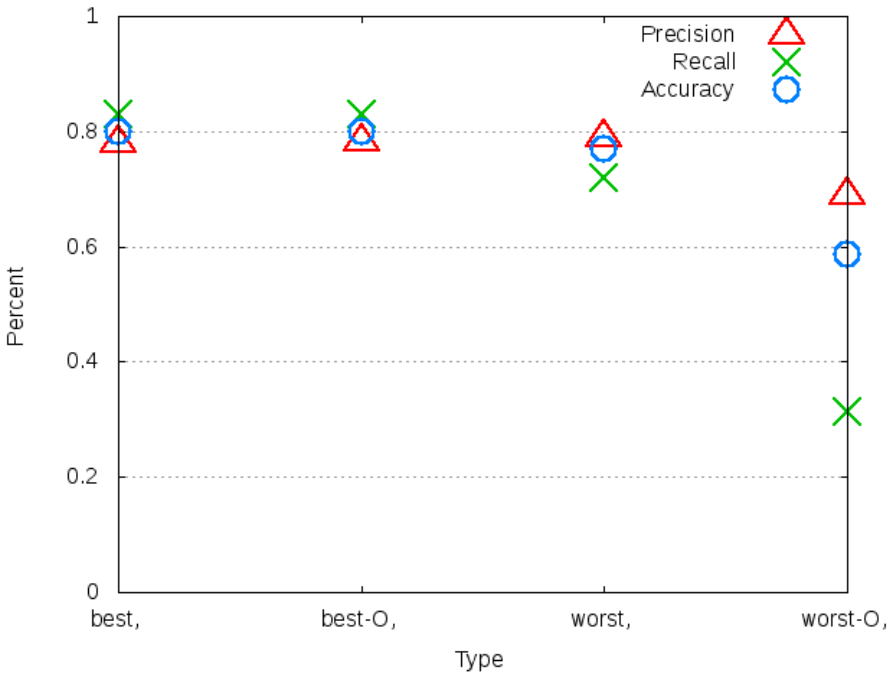


Figure A.156: Oversampling for ShowcaseView using RF

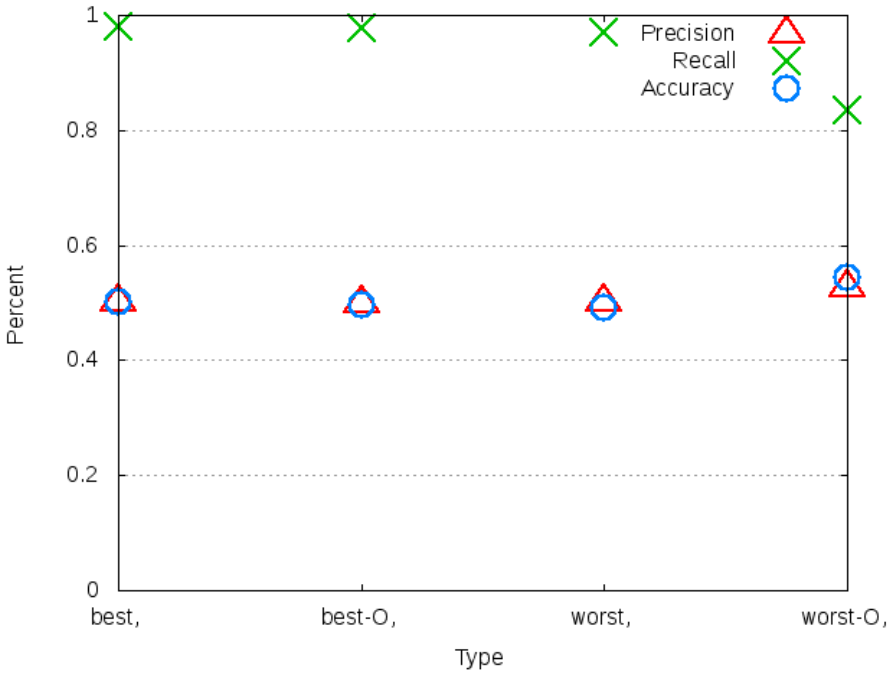


Figure A.157: Oversampling for smile using RF

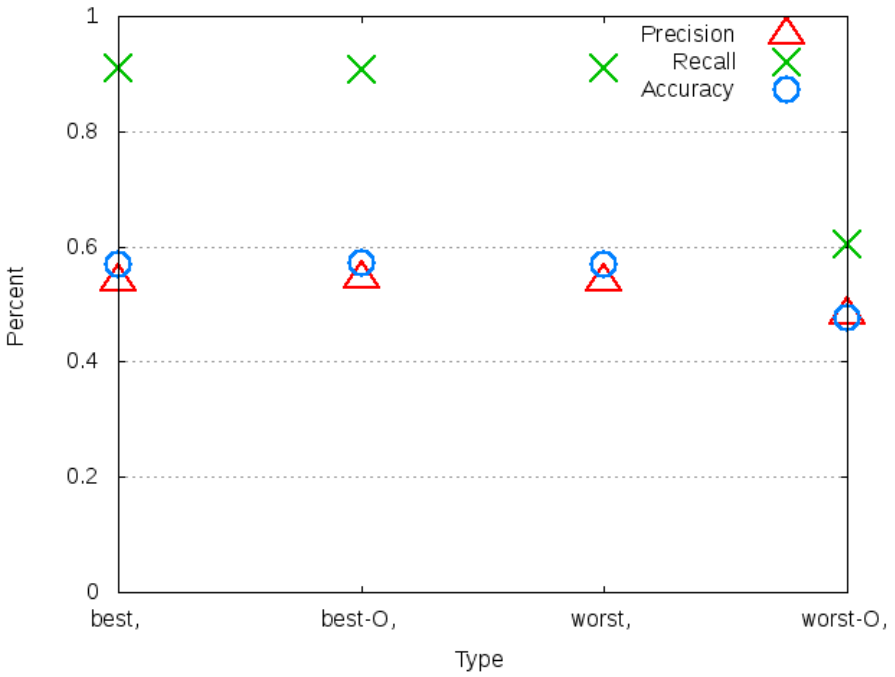


Figure A.158: Oversampling for spark using RF

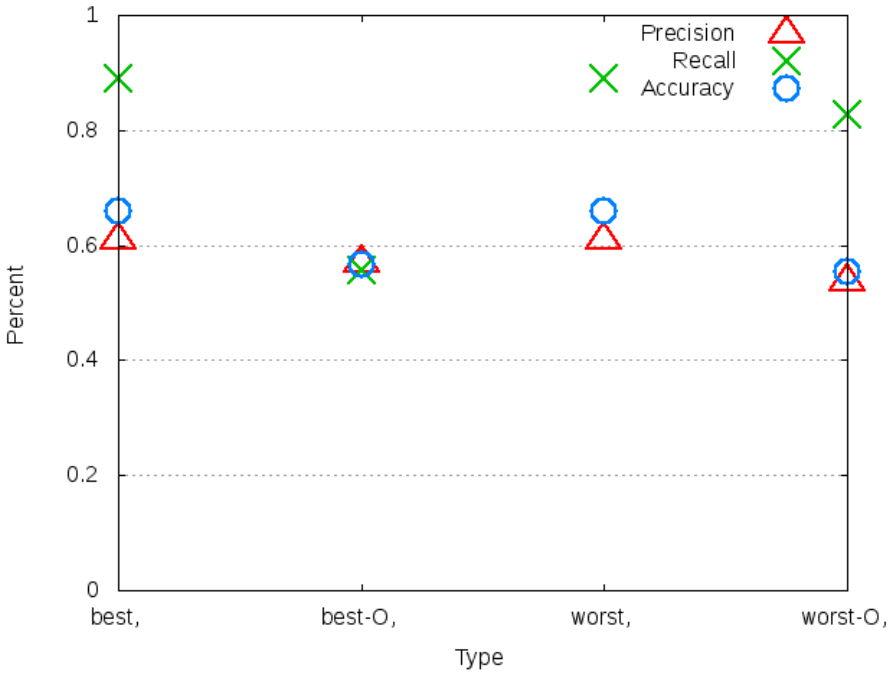


Figure A.159: Oversampling for storm using RF

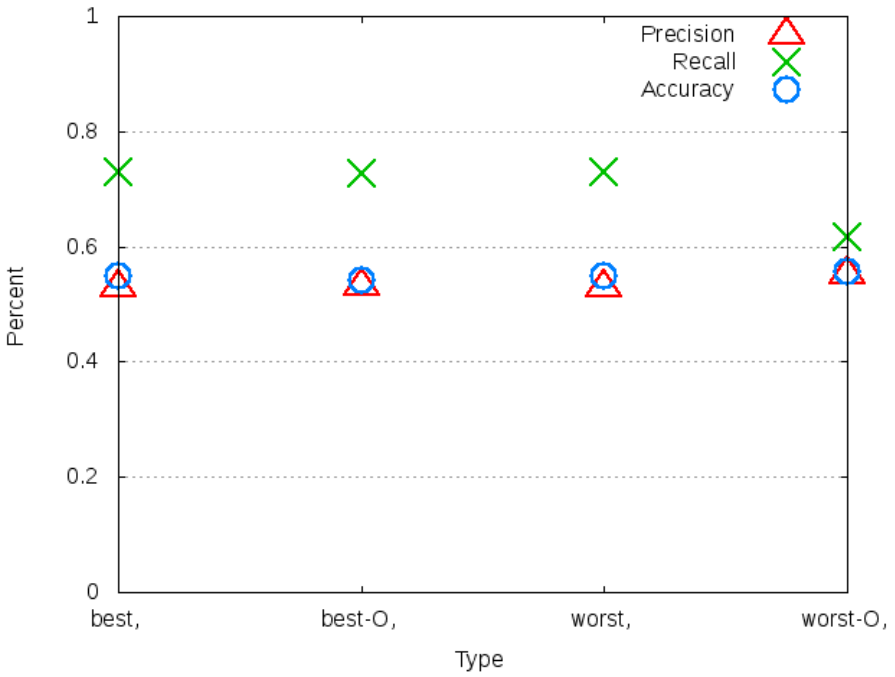


Figure A.160: Oversampling for tempto using RF

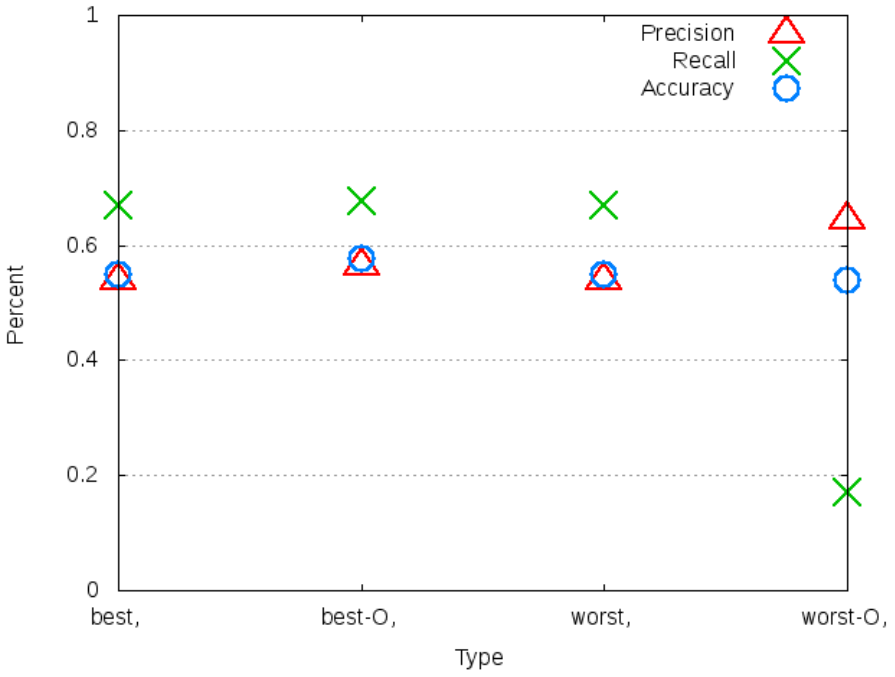


Figure A.161: Oversampling for yardstick using RF