Predicting Mutation Score Using Source Code and Test Suite Metrics

by

Kevin Jalbert

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

Masters of Science

in

Computer Science

University of Ontario Institute of Technology

Supervisor: Dr. Jeremy S. Bradbury

September 2012

Abstract

Mutation testing has traditionally been used to evaluate the effectiveness of test suites and provide confidence in the testing process. Mutation testing involves the creation of many versions of a program each with a single syntactic fault. A test suite is evaluated against these program versions (i.e., mutants) in order to determine the percentage of mutants a test suite is able to identify (i.e., mutation score). A major drawback of mutation testing is that even a small program may yield thousands of mutants and can potentially make the process cost prohibitive. To improve the performance and reduce the cost of mutation testing, we proposed a machine learning approach to predict mutation score based on a combination of source code and test suite metrics. We conducted an empirical evaluation of our approach to evaluated its effectiveness using eight open source software systems.

Keywords: machine learning, mutation testing, software metrics, support vector machine, test suite effectiveness

Co–Authorship

A preliminary version of the research work described in this thesis was previously published in the proceedings of the Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE 2012) [JB12]. I was the primary author of this paper and the research work was conducted under the supervision and in collaboration with Jeremy S. Bradbury.

Acknowledgments

I would like to thank the Ontario Graduate Scholarship (OGS) program for their financial support. I would like to thank UOIT for providing an excellent academic and research environment for the past years I have been a student here.

I would like to thank my supervisor, Jeremy S. Bradbury, for his guidance and wisdom throughout this endeavour. He has been a great supervisor, role-model and colleague. I would like to thank my thesis committee for their insightful feedback and useful comments.

I would like to thank my friends and colleagues within UOIT for being there when a break was needed and for brainstorming ideas. I would like to thank my friends outside of UOIT for being understanding and tolerating the unusual work hours I undertook for research.

I would like to thank my family for their continuous love. My family has always supported every decisions I made, and for that I am grateful. Finally, I would like to thank my fiancé, Katherine Riess, for her love and support over the last two years and many more years to come.

Contents

List of Figures

List of Tables

Abbreviations

CFS Correlation Based Feature Selection.

CLI Command-Line Interface.

CSV Comma Separated Values.

FN False Negative.

FP False Positive.

RBF Radial Basis Function.

SLOC Source Lines of Code.

SUT System Under Test.

SVM Support Vector Machine.

TN True Negative.

TP True Positive.

XML Extensible Markup Language.

Chapter 1

Introduction

1.1 Motivation

A large branch of Software Engineering is software testing and verification. In 2002, a survey showed that inadequate software testing cost the United States approximately $59.5 billion annually [Res02]. This indicates a need to optimize the effectiveness and efficiency of software testing and verification in Software Engineering.

From a software testing perspective, the core artifacts of software development are the source code and the test suite. The source code is composed of many source code units (i.e., methods, classes, functions) while the test suite is composed of many test code units (i.e., test cases and unit tests). As software systems mature over-time these artifacts evolve. For example, during the software development life cycle, specific modules and source code units change to accommodate new features or fixes. If source code units change during development the accompanying tests must also be updated to ensure that the change is adequately tested. Software developers have a number of software testing methodologies and approaches at their disposal to verify the source code of software systems. An essential aspect of most software testing methodologies

is *unit testing* – a white-box testing technique that evaluates source code units to ensure they behave correctly when test code units are applied.

A major challenge in software testing is the assessment of test suites and determining if a given test suite is effective. An effective test suite is *"... one that is capable of detecting all real bugs"* [Wey93] and the purpose of a test suite is to increase confidence that out source code functions correctly. Several techniques exist that measure code coverage (e.g., branch, statement, path) being exercised by a test suite [ZHM97]. Developers are able to assess that the source code units are being tested using one of the coverage criteria provided by a code coverage technique. Unfortunately, simple code coverage might not be an adequate indicator of test suite effectiveness depending on the technique and coverage criteria used [NA09, GJ08].

One approach to determine the effectiveness of a test suite is to use *mutation testing* – a white box coverage technique that assesses the ability of tests to detect *mutant* faults. Specifically, mutation testing uses a set of *mutation operators* to generate faulty versions of a software system's source code called *mutants*. Mutation operators are created based on an existing fault taxonomy and each operator usually corresponds to a specific type of fault. Andrews et al. showed that mutants potentially could be used as substitutes for real faults [ABLN06]. A test suite is evaluated against a set of mutants to determine the *mutation score*. The mutation score is defined as the percentage of non-equivalent mutants that are detected (i.e., *killed*) by a test suite. The better a test suite, the more mutants will be killed and thus the higher the mutation score.

A major drawback of mutation testing is that even a small software system may yield hundreds or thousands of mutants, potentially making the process cost prohibitive in comparison to other coverage metrics. For example, one study produced approximately

2000 mutants for a 5000 Source Lines of Code (SLOC) software system (`jtopas`) and approximately 105000 mutants for a 30000 SLOC software system (`xstream`) [SZ09a].

1.2 Problem

Mutation testing offers a highly effective approach for determining the effectiveness of a test suite but at high cost. The adoption of mutation testing in industry has been slow due to the performance/scalability issues and tool usability (i.e., integration into a standard software development life cycle) [OU01]. Three approaches have been proposed to improve mutation testing performance and scalability [OU01]:

1. **"Do fewer" approach:** This category of optimizations aims to decrease the computational cost of mutation testing by reducing the number of mutants that a test suite is evaluated against. The most popular example from this category is selective mutation – the use of a subset of mutation operators that have been empirically shown to be as effective as using an entire set of operators [OLR+96].

2. **"Do smarter" approach:** This category of optimizations aims to decrease the cost of mutation testing by improving the actual mutation testing technique. For example, weak mutation *"... is an approximation technique that compares the internal states of mutant and original program immediately after execution of the mutated portion of the code (instead of comparing the program output)"* [OU01].

3. **"Do faster" approach:** This category of optimizations aims to reduce the cost of mutation testing by focusing on performance. For example, one "do faster" approach improves compilation time using schema-based mutation – *"... encodes all mutations into one source level program ..."* [OU01].

As an alternative to the above approaches, we propose a *"do fewer and smarter"* approach for mutation testing at the unit level. When mutation testing is used for the creation or improvement of a test suite, the test suite will often have to be applied to the mutants in an iterative fashion as tests are added, removed and modified. Furthermore, the effects on the mutation score after each iteration have to be observed. We propose to replace at least some of the mutation testing with mutation score prediction and thus decrease the number of mutants that have to be evaluated using a test suite. Our proposed approach uses machine learning to predict the mutation score based on a combination of source code and test suite metrics of the code unit under test.

1.3 Thesis Statement and Scope of Research

> ***Thesis Statement:*** *The use of source code and test suite metrics in combination with machine learning techniques can accurately predict mutation scores. Furthermore, the predictions can be used to reduce the performance cost of mutation testing when used to iteratively develop test suites.*

Essentially, this thesis presents an approach that predicts the mutation scores of code units. This approach is ideal for the iterative creation or improvement of a test suite as it mitigates the amount of time spent on mutation testing (i.e., less testing of mutants).

The scope of this thesis is limited to open source Java software systems. We have selected this scope because mutation testing of Java software systems is fairly mature and there are a number of existing mutation tools for Java [JH11].

1.4 Contributions

This thesis makes the following contributions to the field of mutation testing, software testing and software quality assurance:

- An approach to predict the mutation scores of source code units of software systems using a machine learning technique.
- Identify source code and test suite metrics that are capable of describing source code units with respect to mutation score prediction.
- An empirical evaluation of the accuracy of the developed approach with respect to the identified source code and test suite metrics.
- An empirical evaluation of the accuracy of the developed approach with respect to prediction of unknown data within a software system and across software systems.
- Identify a generalizable set of parameters for our machine learning technique to maximize prediction performance over different software systems.
- Demonstrate that traditional training/testing data ratios are not necessary to achieve near optimal prediction performance of mutation scores.

1.5 Organization of Thesis

In this chapter we have outlined our motivation in Section 1.1 and the problem in Section 1.2. We presented our thesis statement in Section 1.3 along with a general set of contributions for this thesis in Section 1.4. The remaining chapters of this thesis are organized as follows:

- *Chapter 2:* We describe the background material and concepts used in this thesis. We look at uses and details of machine learning on how it can be used for classification problems. We illustrate what mutation testing is and the advantages and disadvantages of using it. We finally cover software metrics and their uses in understanding software systems and complexity.

- *Chapter 3:* We describe our overall approach to mutation score prediction. We cover each step of our approach all while detailing the selected tools used. We discuss related work in the area of predictions using source code metrics.

- *Chapter 4:* We describe our experimental setup as well as our eight selected test subjects. We conduct a number of experiments to evaluate our approach along with discussions.

- *Chapter 5:* We summarize the thesis and mentioned the contributions. We outline future work as well as limitations on the thesis.

Chapter 2

Background

In this chapter we describe the background techniques and tools used in our research. Specifically, we cover mutation testing in Section 2.1, machine learning in Section 2.2 and software metrics in Section 2.3.

2.1 Mutation Testing

As mentioned in Section 1.1, techniques exists that measure code coverage. Mutation testing can be seen as a fault-based coverage technique that demonstrates the absence of faults in a software system [DLS78, BDLS80]. Mutation testing makes use of fault-based testing by generating a set of mutants, each representing a possible fault in the software system. These mutants are then executed along with the test suite with hopes that the test suite can detect the mutant's fault. If the fault is detected, the test suite is effective enough to handle the detection of that specific bug. If the fault goes undetected, the test suite ability to detect that specific bug is inadequate. Using the results of mutation testing, it is possible to assess the adequacy of a test suite – the effectiveness of the test suite of detecting bugs.

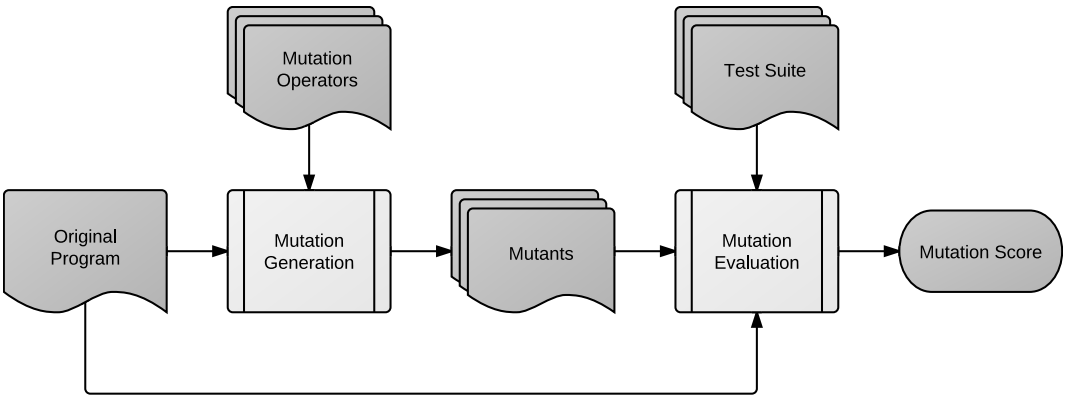


Figure 2.1: The mutation testing process.

Figure 2.1 illustrates a general approach to mutation testing. Mutation testing uses a set of *mutation operators* to generate faulty versions of a software system's source code called *mutants*. A mutation operator applies a transformation to a software artifact such that it now exhibits a fault (see Section 2.1.1.1 for examples). Mutation operators are designed based on existing fault taxonomy, such that the generate mutants represent real faults. Studies have indicated that mutants could be used as substitutes for real faults [ABLN06, ABL05, NK11].

The transformation of a software artifact to create a mutant is typically a small/single change as most bugs follow the *Competent Programmer Hypothesis* [ABD+79] which suggests that developers write software that is nearly correct. Also the *Coupling Effect Hypothesis* [Off92] suggests that a large percent of complex faults can be detected if all the simple faults can be detected. These two hypotheses strengthen the use of small/single changes for mutation operators and why mutation testing is adequate for evaluating test suite effectiveness.

$$mutation\ score = \frac{killed\ mutants}{total\ mutants - equivalent\ mutants} \tag{2.1}$$

After all the mutants have been generated, a testing approach is used to evaluate the mutants against the test suite. If a mutant is detected by the test suite, the mutant is *killed*. If undetected, we say it *survived*. There are some cases where the mutant generated is *equivalent*, such that the behaviour of the mutant is the same as the original system. These equivalent mutants pose a problem as they cannot be killed using the given test suite. Manual inspection of mutants to determine if they are equivalent is not feasible for a large number of mutants. A *mutation score* (see Equation 2.1) is given to each source code unit based on the number of percentage of non-equivalent mutants they killed. The mutation score indicates how effective a test suite is at detecting faults in terms of mutation fault-based testing adequacy.

Mutation testing has traditionally been used as a coverage technique to evaluate the effectiveness of test suites and provide confidence in the testing process [JH11]. For over 30 years, mutation testing has been applied to software written in programming languages including C [DM96, JH08], Fortran [KO91] and Java [MKO02, BCD06]. Furthermore, mutation testing has also been applied to non-programming artifacts such as formal specification languages [ABM98], markup languages [PO10] and spreadsheets [AE09].

The following discussion presents the two major criticisms of mutation testing, accompanied with some of the research has been done to alleviate these to some degree:

- **Equivalent Mutants:** As already described, these are mutants that are semantically the same as the original version of the software system. An equivalent mutant will not be killed by the test suite and this results in lower than expected mutation scores. These are problematic as mutation score is influenced by these mutants though they are difficult/costly to detect. Schuler and Zeller proposes a solution in determining whether a surviving mutant is equivalent or

not using impact analysis [SZ10]. Their approach observes the impact of the original program's execution against that of the mutant in respect to control flow and data. Their experiments showed that using statement coverage allowed them to achieved a classification precision of 75% and a recall of 56%. In their previous work they also considered the use of the impact of dynamic invariants to uncover equivalent mutants [SDZ09]. Offutt and Craft used compiler optimization techniques and were able to detect approximately 10% of equivalent mutations [OC94].

- **Performance Cost:** Again as we have already mentioned mutation testing is a very costly coverage technique as many mutants must be evaluated against the test suite. The mutant representation, selection of tests, and strategies are all aspects of the mutation testing process that the research community are exploring to reduce cost. Mutation sampling can be used to reduce the evaluation efforts by only considering a random subset of the generated mutants [Bud80]. Untch et al. introduced a mutation runtime technique call *Mutant Schema Generation* that represented all possible mutants in a single meta-program [UOH93]. Offutt et al. were able to perform mutation testing at the bytecode level, effectively avoiding recompilations of the generated mutants [OMK04].

2.1.1 Mutation Operators

As previously mentioned, mutation operators define transformations that attempts to introduce faults. Focusing on Java, there are two common sets of mutation operators: method-level and class-level. These two sets of mutation operators are described in the following sections.

Operator	Description
AOR	Arithmetic Operator Replacement
AOI	Arithmetic Operator Insertion
AOD	Arithmetic Operator Deletion
ROR	Relational Operator Replacement
COR	Conditional Operator Replacement
COI	Conditional Operator Insertion
COD	Conditional Operator Deletion
SOR	Shift Operator Replacement
LOR	Logical Operator Replacement
LOI	Logical Operator Insertion
LOD	Logical Operator Deletion
ASR	Assignment Operator Replacement

Table 2.1: The set of method-level mutation operators from the *MuJava* mutation testing tool [MOK05, MO05b].

Original Program

```
class Counter {
  Integer current = new Integer(1);
  Integer limit = new Integer(10);
  public Integer add() throws Exception {
    if (this.current > this.limit) {
      throw new Exception();
    }
    return ++this.current;
  }
}
```

$\xrightarrow{ROR}$

Mutant Program

```
class Counter {
  Integer current = new Integer(1);
  Integer limit = new Integer(10);
  public Integer add() throws Exception {
    if (this.current < this.limit) {
      throw new Exception();
    }
    return ++this.current;
  }
}
```

Figure 2.2: Example application of the *ROR* method-level mutation operator.

2.1.1.1 Method-Level Mutation Operators

We first consider the set of method-level mutation operators found in the mutation testing tool *MuJava* [MOK05], as they are well documented and designed. These mutation operators apply source transformations that modify expressions at the method-level. These operators can cause unexpected data values to occur, as well as adjust the outcome of conditions. A set of method-level mutation operators are listed in Figure 2.1 [MO05b].

Original Program

```
class Counter {
  Integer current = new Integer(1);
  Integer limit = new Integer(10);
  public Integer add() throws Exception {
    if (this.current > this.limit) {
      throw new Exception();
    }
    return ++this.current;
  }
}
```

$\xrightarrow{AOI}$

Mutant Program

```
class Counter {
  Integer current = new Integer(1);
  Integer limit = new Integer(10);
  public Integer add() throws Exception {
    if (this.current > this.limit--) {
      throw new Exception();
    }
    return ++this.current;
  }
}
```

Figure 2.3: Example application of the *AOI* method-level mutation operator.

To illustrate the effects of a method-level operator, consider the *Relational Operator Relational* (*ROR*) mutation operator. This mutation operator replaces a relational operator (i.e., `>`, `>=`, `==`, `!=`, `=<` or `<`) with another type of relational operator as seen in Figure 2.2. Figure 2.3 presents another example demonstrating the *Arithmetic Operator Insertion* (*AOI*) mutation operator. The remaining set of method-level mutation operators function using a similar approach with other operators (i.e., conditional, shift, logical and assignment).

2.1.1.2 Class-Level Mutation Operators

We now look at the set of class-level mutation operators found in *MuJava* [MOK05, MKO02]. These mutation operators apply source transformations that modify language features at the class-level. These operators can allow objects to behave in unexpected ways, as well as exposing design issues. Table 2.2 tabulates the class-level mutation operators [MO05a].

To illustrate the effects of a class-level operator, we can look at the *Member Variable Initialization Deletion* (*JID*) mutation operator. This mutation operator deletes an instance variables initialization as sees in Figure 2.4. Figure 2.5 presents another example demonstrating the *Access Modifier Change* (*AMC*) mutation operator. The remaining set of class-level mutation operators function by inserting, deleting, and

Group	Operator	Description
①	*AMC*	Access modifier change
②	*IHD*	Hiding variable deletion
②	*IHI*	Hiding variable insertion
②	*IOD*	Overriding method deletion
②	*IOP*	Overriding method calling position change
②	*IOR*	Overriding method rename
②	*ISI*	`super` keyword insertion
②	*ISD*	`super` keyword deletion
②	*IPC*	Explicit call to a parent's constructor deletion
③	*PNC*	`new` method call with child class type
③	*PMD*	Member variable declaration with parent class type
③	*PPD*	Parameter variable declaration with child class type
③	*PCI*	Type cast operator insertion
③	*PCC*	Cast type change
③	*PCD*	Type cast operator deletion
③	*PRV*	Reference assignment with other comparable variable
③	*OMR*	Overloading method contents replace
③	*OMD*	Overloading method deletion
③	*OAC*	Arguments of overloading method call change
④	*JTI*	`this` keyword insertion
④	*JTD*	`this` keyword deletion
④	*JSI*	`static` modifier insertion
④	*JSD*	`static` modifier deletion
④	*JID*	Member variable initialization deletion
④	*JDC*	Java-supported default constructor creation
④	*EOA*	Reference assignment and content assignment replacement
④	*EOC*	Reference comparison and content comparison replacement
④	*EAM*	Acessor method change
④	*EMM*	Modifier method change

Table 2.2: The set of class-level mutation operators from the *MuJava* mutation testing tool [MOK05, MO05a].

The group column indicates the specific language feature of the mutation operator (①: Encapsulation, ②: Inheritance, ③: Polymorphism, ④: Java-Specific Features).

changing certain elements in the class with respect to inheritance, polymorphism, and Java-specific features.

Original Program

```
class Counter {
  Integer current = new Integer(1);
  Integer limit = new Integer(10);
  public Integer add() throws Exception {
    if (this.current > this.limit) {
      throw new Exception();
    }
    return ++this.current;
  }
}
```

$\xrightarrow{JID}$

Mutant Program

```
class Counter {
  Integer current = new Integer(1);
  Integer limit;
  public Integer add() throws Exception {
    if (this.current > this.limit) {
      throw new Exception();
    }
    return ++this.current;
  }
}
```

Figure 2.4: Example application of the *JID* class-level mutation operator.

Original Program

```
class Counter {
  Integer current = new Integer(1);
  Integer limit = new Integer(10);
  public Integer add() throws Exception {
    if (this.current > this.limit) {
      throw new Exception();
    }
    return ++this.current;
  }
}
```

$\xrightarrow{AMC}$

Mutant Program

```
class Counter {
  Integer current = new Integer(1);
  Integer limit = new Integer(10);
  private Integer add() throws Exception {
    if (this.current > this.limit) {
      throw new Exception();
    }
    return ++this.current;
  }
}
```

Figure 2.5: Example application of the *AMC* class-level mutation operator.

2.1.1.3 Other Mutation Operators

In addition to the two general sets of mutation operators just described, sets also exist for specific domains (i.e., concurrency and security). Bradbury et al. presented a set of concurrency mutation operators that is capable of creating concurrency faults (e.g., data races and deadlocks) [BCD06]. Shahrair and Zulkernine presented multiple sets of mutation operators in the security domain for database injection [SZ08a], buffer overflows [SZ08b], and cross site scripting [SZ09b]. Offutt et al. presented operators for grammar-based testing [OAL06]. Furthermore, as mentioned earlier on in Section 2.1, other domains where mutation testing has been applied have their own set of mutation operators (i.e., spreadsheets, markup), however this is outside the scope of this thesis.

2.1.2 Mutation Testing Tools

In the last decade, a number of mutation testing tools for the Java programming language have emerged [JH11]. We present the following set of criteria that distinguishes most of the tools from each other:

- **Citation**: The citation for the tool's publication/website.
- **Inception Year**: The year the tool was released to the public.
- **Generation-Level**: Mutations can be generated either at the source code or bytecode level. Source code mutation generation requires re-compilation while bytecode does not.
- **Test Selection**: Test selection indicates which unit test cases are performed for each mutant. A naive approach simply runs all the unit test cases, while convention based runs all tests based on a package/test name or defined annotations. Coverage based approach only runs unit test cases that are directly involved in the mutated source code, while a manual bases approach allows the user to specify each unit test case.
- **Mutant Insertion**: Generated mutants are stored and ran against the selected unit test cases. A naive approach stores the mutants on disk and creates a new Java Virtual Machine for each mutant. A schmeta approach stores all the mutants in a single class, and the mutants are enabled through runtime flags one at a time [UOH93]. An in-memory approach stores all the mutants in memory which are then injected into the Java Virtual Machine by creating a new classloader. An instrumentation approach stores the mutants in memory, but injects them into the Java Virtual Machine directly using an instrumentation application programming interface.

- **Method-Level**: Whether or not a tool has a set of *traditional* method-level mutation operators, as mentioned in Section 2.1.1.1.
- **Class-Level**: Whether or not a tool has a set of object-oriented class-level mutation operators, as mentioned in Section 2.1.1.2.
- **JUnit Support**: Whether or not a tool has support for JUnit test cases (de facto for unit testing Java [Bec].
- **Command-Line**: Whether or not a tool has support to be executed via a Command-Line Interface (CLI).
- **Structured Output**: Whether or not a tool has support to output results in a structure format (i.e., Extensible Markup Language (XML), Comma Separated Values (CSV))
- **Unit Scores**: Whether or not a tool indicates the mutation score of individual source code units (i.e., the mutation score of methods).
- **Open Source**: Whether or not a tool's source code is open source and freely available to modify.
- **Academic Tool**: Whether or not a tool was developed from an academic research group, otherwise industry or community developed.
- **Special Feature**: Whether or not a tools has a special feature that is unique in mutation testing.

Using this criteria, we provide a comparison of a set of mutation testing tools for the Java programming language in Table 2.3. We can see various aspects of the mutation testing tools. For example, only *Javalanche* has access to a limited set of

Table 2.3: A categorization of several Java mutation testing tools and their feature (compiled using data from [Col, MR10]).

	Jester	***MuJava***	***Jumble***	***Javalanche***	***Judy***	***PIT***
Citation	[Moo]	[MOK05]	[Jum]	[SZ09a]	[MR10]	[Col]
Inception Year	2001	2004	2007	2009	2010	2011
Generation-Level	Source Code	Bytecode	Bytecode	Bytecode	Source Code	Bytecode
Test Selection	Naive[f]	Manual	Convention	Coverage	Convention	Coverage
Mutant Insertion	Naive[f]	Schmeta	In-Memory	Schmeta	Schmeta[c]	Instrument
Method-Level	No	Yes	No	Yes	Yes	Yes
Class-Level	No	Yes	No	No	Yes[b]	No
JUnit Support	Yes	No[c]	Yes	Yes	Yes	Yes
Command-Line	Yes	No	Yes	Yes	Yes	Yes
Structured Output	No	No	No	Yes	No	Yes
Unit Scores	No	No	No	Yes	No	No[g]
Open Source	Yes	No[d]	Yes	Yes	No	Yes
Academic Tool	No	Yes	No	Yes	Yes	No
Special Feature	No	No	No	Yes [eh]	No	No

[a] The Eclipse plugin has support for JUnit 3.
[b] Limited set of class-level operators.
[c] Iterative schmata (only allows a certain limit of mutants per iteration).
[d] Limited to researchers by request.
[e] Impact analysis to deal with equivalent mutants.
[f] Best guess considering limited documentation.
[g] The unit score could be calculated manually using structured output.
[h] Limited set of concurrency mutation operators.

concurrency-level mutation operators while *Jumble* and *Jester* do not use a proper form of the method- and class-level mutation operators.

2.2 Machine Learning

Machine learning is a branch of artificial intelligence that primarily focuses on the ability to classify complex data. Given a data set with complex relationships, machine learning algorithms can attempt to uncover patterns that characterize the data. With the uncovered patterns and relationships it is then possible to make intelligent predictions based on the data. For example, given historic data about the weather (i.e., rain, wind speed, pressure, humidity, etc...) we could use machine learning techniques to make a prediction on whether it will rain or not tomorrow.

Machine learning algorithms are either a *supervised* or *unsupervised* learning algorithm. The main difference between the two types of algorithms is whether one can correctly classify data prior to using the algorithm. In situations where no data classification information is known about the data set, then a unsupervised classification algorithm could be used. Unsupervised learning algorithms aim to classify the data based on density or clusters (e.g., k-mean clustering). Supervised learning requires that an *expert* can classify some of the data to use for *training*. A supervised learning algorithm creates a model that describes the training data. Unclassified data can then be passed through the model to see what classification best fits the data.

If one does not have any idea of what they are trying to classify, unsupervised learning is the best approach. It is not possible to categorize the data correctly, and incorrect categorization would only be detrimental to the classification. If it is possible to categorize the data, it becomes possible to perform supervised learning on the data.

With correct categorization for the training data, future data can be classified on the constructed model from the training phase.

In either case, both learning algorithms require data to be classified. Each element within the data set consists of a feature set of size n. The power of machine learning comes from its ability to handle multiple dimensions of features for multiple dimensions of classification categories. For unsupervised learning, each element within the data set does not require a classification category, however for supervised learning this pre-determined category is required (at least initially to build the model).

Many areas of research have benefited from machine learning, such as data mining [WFH11], computer vision [Her03], biology [OLP08], business [Her00], and health sciences [Kon01]. Furthermore, most companies that offer a complex recommendation system utilize machine learning techniques. For example, in 2006 *Netflix* challenged the computer science, data mining, and machine learning communities to develop a system that exceeded its own recommendation system [BL07]. Several tools exist that provide sophisticated machine learning techniques to the general community in an off-the-shelf toolkit format such as *WEKA* [HFH+09] and *SHOGUN* [SRH+10].

2.2.1 Performance Measures

Machine learning techniques can be extremely beneficial in solving classification problems. To properly measure the performance of the classification, the actual *known* classification is required for each data instance. For unsupervised learning, the performance measures can be acquired immediately after the data has been classified, as the correct classifications are present. In supervised learning, we first need to generate a model that we can use to classify new data. To accommodate measuring supervised learning, the available data is split up into *training* and *testing* sets. The training set is used to construct the model which is then used to predict the testing

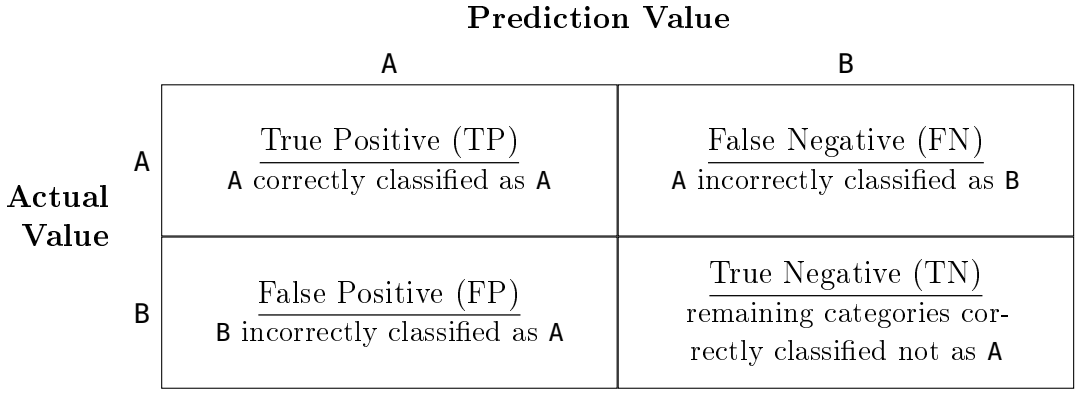


Figure 2.6: A 2 $\times$ 2 confusion matrix for classification results of the A category.
It is possible to extend a confusion matrix to $n \times n$ dimensions. For each category the (TP, FN, FP, and TN) variables need to be calculated.

set. The classifications of the predicted set can then be verified against their correct classification to acquire performance measures of the supervised learning technique. It is also possible to perform *cross-validation*, which is a technique that assess the prediction accuracy using only the trained data. This evaluation is typically used when only a small amount of data is available for training. Effectively, this randomly divides the training data into n equal-sized partitions. The machine learning technique than trains on $n - 1$ partitions and predicts on the last. This process of training and prediction is repeated n times using a different partition for prediction each time. Finally, all the individual prediction accuracies are tallied and averaged for a cross-validation accuracy of n-folds. Commonly a 10-fold cross-validation is used to evaluate predictive models [Koh95].

A confusion matrix can visually represent the allotment of predictions over the actual categories, as seen in Figure 2.6. Using the variables of a confusion matrix, it is possible to construct several performance measures that describe the effectiveness of

the classifier that created the confusion matrix. The following performance measures are commonly used to describe the performance of a classifier [1] [SJS06]:

- **Precision** represents the fraction of positive predictions that correctly belong to the positive category.

$$precision = \frac{TP}{TP + FP} \tag{2.2}$$

- **Recall** represents the fraction of positive instances that were correctly identified.

$$recall = \frac{TP}{TP + FN} \tag{2.3}$$

- **Specificity** represents the fraction of negative predictions that were correctly identified.

$$specificity = \frac{TN}{TN + FP} \tag{2.4}$$

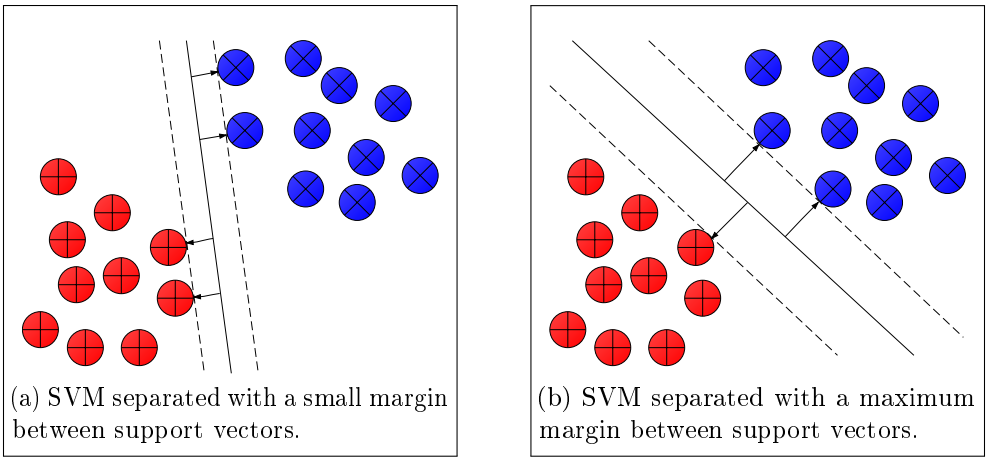
- **Accuracy** represents the fraction of all true predictions made that were correctly identified.

$$accuracy = \frac{TP + TN}{TP + FP + FN + TN} \tag{2.5}$$

2.2.2 Support Vector Machine

A Support Vector Machine (SVM) is an example of a linear discrimination machine learning technique and assumes that *"... instances of a class are linearly separable from instances of other classes"* [Alp04]. Traditionally, SVMs have been used for two-group classification problems [CV95], but have also been generalized to n-group classification problems. For example, a SVM could be used to distinguish source code

[1] All of these measures can all be applied to each category. For demonstrative purposes we focus on the positive category.



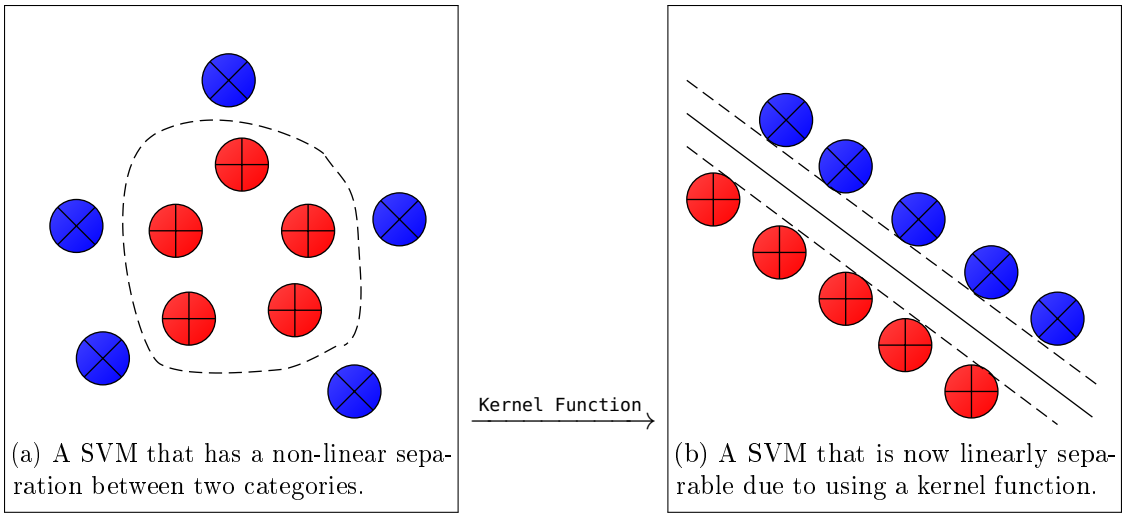
(a) SVM separated with a small margin between support vectors.

(b) SVM separated with a maximum margin between support vectors.

Figure 2.7: Difference between small and maximum margins between support vectors. *As we can see in the two above examples, there are two categories in the feature space with a solid line separating them, this is called the 'hyperplane'. The dotted lines represent the distance to the closest vector from the hyperplane, the distance between both dotted lines is also called the 'margin'. 'Support vectors' are the vectors that are touching the margin. The goal of a SVM is to maximize the distance between support vectors of opposite categories using the hyperplane. There is a clear difference in the distance separating support vectors in the above examples, with the maximum margin (b) being a better solution then the small margin (a).*

written by two developer. To perform this classification a set of attributes are required to construct the SVM *feature space*. A *feature space* consists of a set of *vectors* (i.e., a row of data in the data set), with each vector containing a set of *attributes*[2] (i.e., the values that define the vector). Many attributes can be used to aid the SVM in distinguishing between the different categories. Using our example, we might consider the lines of code, comment ratio, test coverage amongst other software metrics as attributes. In our example the category for each vector then represents the developer for the source code. With vectors consisting of attributes and category information the SVM should be able to distinguish the source code of the two developers based on the attributes of the source code.

[2]With respect to our approach, the use of *attributes*, *metrics* and *features* are interchangeable. Using our example these attributes refer to the value of a metric that is measured from the software system.



(a) A SVM that has a non-linear separation between two categories.

(b) A SVM that is now linearly separable due to using a kernel function.

Figure 2.8: Linearly separating non-linear using a kernel function.

As we can see in the two above example, a non-linear hyperplane is required in (a) to properly separate the two categories. SVMs attempt to linearly separate the feature space. In this case it is possible to map the vectors to a higher dimension using a kernel function. *In the above example a kernel function is used to map (a) to (b), resulting in a higher dimension such that each vector now has a radius value (distance from the center to the edge). As we can see in (b) it is now linearly separable when the feature space is mapped to a higher dimension (from two-dimensions to three-dimensions).*

To further illustrate how a SVM works we present an example (see Figure 2.7), in which each vector has two attributes (x and y coordinates) and a category (blue or red). A SVM attempts to find the maximum margin space between support vectors of opposite categories, which results in the optimal hyperplane. The optimal hyperplane is chosen over others, because *"Intuitively, we would expect this boundary to generalize well as opposed to the other possible boundaries"* [Gun98].

In some cases a linear separation is not possible in the feature space. An example of this is presented in Figure 2.8. It is possible to map the vectors to a higher dimension, so that the feature space now becomes linearly separable. The separating hyperplane is always of $n-1$ dimensions (e.g., two dimensional data is separated with a one dimensional line). Several kernel functions exist, though the *Radial Basis Function (RBF)* kernel is highly recommended by the authors of *LIBSVM* [HCL03]. Kernels

have parameters that govern how they form their corresponding hyperplane. The RBF kernel has a *gamma* parameter that governs the flexibility (i.e., curvature) of the hyperplane. If the hyperplane is too flexible (i.e., follows the contour of the data too strictly) then it runs the risk of being overfitted for the given data [BHW10]. Overfitting can lower the ability of a classifier to generalize. SVMs address this by maximizing the margin distance, which allows some flexibility in adding new vectors to the feature space. SVMs also have a *cost* parameter, where if there is a low cost then the SVM will allow some mis-classifications (within a distance from the hyperplane using a function of cost) [BHW10]. A higher cost value will reduce the number of mis-classifications, but may create a model that does not generalize outside of the training data.

Considering the following criteria for SVMs, we can distinguish between several different implementations from a user perspective:

- **Citation**: The citation for the tool's publication/website.
- **Inception Year**: The year the tool was released to the public.
- **Multiple-Group**: Whether or not the tool supports multiple group classification problems, in addition to binary group classification.
- **Cross-Validation**: Whether or not the tool supports cross-validation.
- **Measures**: Whether or not the tools supports performance measures to aid in evaluation of classification accuracy.
- **Command-Line**: Whether or not the tool supports to be ran via a CLI.
- **Open Source**: Whether or not the tool's source code is open source and freely available to modify.

	SVM^{light}	***LS-SVMlab***	***LIBSVM***
Citation	[Joa99]	[SV99,PSVG^{+}02]	[CL11]
Inception Year	1999	1999	2000
Multiple-Group	No[a]	Yes	Yes
Cross-Validation	Yes[e]	Yes	Yes
Measures	No[b]	Yes	No[c]
Command-Line	Yes	No[d]	Yes[f]
Open Source	Yes	Yes	Yes
Academic Tool	Yes	Yes	Yes

[a] There is an alternative tool of the same family that allows this ($SVM^{multiclass}$).
[b] There is an alternative tool of the same family that allows this only on binary classification (SVM^{perf}).
[c] There is an external extension which allows this only for binary classification.
[d] Is a MATLAB toolkit, though it is possible to run a MATLAB script via a CLI.
[e] Performs a *Leave-One-Out* cross-validation, which is a n-fold cross-validating given a n vectors in the feature set.

Table 2.4: Basic comparison of different SVM implementations from a user perspective.

- **Academic Tool**: Whether or not the tool was developed from an academic research group, otherwise industry or community developed.

Using this criteria, we provide a comparison of a different SVM implementations in Table 2.4. We can see various aspects of different SVM implementations, for example all three of these SVMs are open source academic tools that support cross-validation. SVM^{light} does not support multiple-group classification, though it has an alternative tool that does allow this. *LS-SVMlab* has build in performance measurements, though the tool works as a MATLAB toolkit.

2.3 Software Metrics

Metrics are measurements of a system, which can provide insight in describing/understanding the system. Goodman defines software metrics as *"The continuous application of measurement-based techniques to the software development process and its products*

to supply meaningful and timely management information, together with the use of those techniques to improve that process and its products" [Goo93]. Measurements can be further defined using the following definitions [Fen94]:

- **Entity:** Represents an object or event.
- **Attribute:** Represents a feature or property of an *entity*.
- **Model:** Represents a specific viewpoint of an *attribute*.

With respect to software metrics, we can consider a multitude of entities such as the source code, the test cases, the bug reports, and more. There are many possible attributes that can be used for any entity, it just has to be a repeatable and measurable property. A measurable attribute is not sufficient as there might be different views on how to interpret the attribute. For example, using a software system's source code as the entity and the size in lines of code being the attribute, how do we view or represent size in this case? Should we include blank lines and/or comments? Are we considering logical lines, physical lines? A model is used to specify the specific viewpoint of the attribute, therefore with respect to the previous question we might view size with respect to physical lines that exclude blank lines and comments.

As mentioned there are a number of software metric entities available. With respect to source code as an entity (see Section 2.3.1), there are a number of attributes that can represent structural characteristics of the source code. With respect to test suite as an entity (see Section 2.3.2), there are attributes related to test coverage, as well source code attributes as a test suite is often just source code that exercise the software system. More software metrics entities exist (e.g., software development life-cycle, bug report(s), program execution [SS08]) that we do not explain in this chapter as it is not required for background knowledge.

2.3.1 Source Code Metrics

In general, software metrics can be used to measure a number of qualities of a software system. In particular, source code metrics give insight into structural aspects of the software system including its complexity, size, and object-oriented attributes [McC76, Kan02, HWY09, HS96, SRD12]. Chidamber and Kemerer presented a suite of object-oriented metrics [CK94], around the same time Abreu and Carapuça also presented the MOOD object-oriented suite [AC94]. Source code metrics are typically extracted from the source code using static analysis techniques. Some metrics like *defect density* make use of external bug reports in combination with the source code to indicate problematic modules [FP98].

For the scope of this thesis, we only consider the Java programming language, which has object-oriented features. This means that source code metrics can be acquired at different scope-levels (i.e., method, class, package, project). For example, we can calculate the cyclomatic complexity [McC76] of a method by simply counting the number of decision points (i.e., different decisions of control statements). We can also calculate the nested block depth of a method, which is determined by the number of nested blocks (i.e., control statements). From a class-level perspective we can measure the number of methods and attributes the class contains. We can also measure the depth of the class in respect to its inheritance tree. At a higher level we start to consider the number of classes within a package, along with measuring coupling inside and outside of the package. These metrics can be used to alert developers to a specific class or method might be problematic. If these metrics reach extreme points (i.e., complex method, large class, high coupling) they can become *code smells* (i.e., source code that is hard to read and maintain) and should be refactored [FB99].

2.3.2 Test Suite Metrics

The source code of a software system is one of the most important software artifacts. From a testing perspective the test suite is very important as it ensures the correctness of the System Under Test (SUT). Using the test suite as an entity for software metrics provides a number of observable attributes. As test cases at a unit testing level are just source code units, it is possible to borrow similar attributes from the source code metrics entity (see Section 2.3.1). As mentioned in Section 1.1, coverage can also assess what parts of the source code are exercised by the test suite [ZHM97]. Coverage metrics is one of the more common test suite metrics, as it measures the relationship between the test suite and source code. We can also extract specific test suite attributes such as the number of test cases, and also attributes from the source code entity such as the complexity of test cases.

2.4 Summary

In this chapter we covered the following background topics for the research presented in this thesis:

- In Section 2.1 we covered what mutation testing is and how it relates to test suite effectiveness with respect to fault detection adequacy. We explored various sets of mutation operators with examples for method- and class-level mutation. A set of Java specific mutation tools was also discussed for comparison.

- In Section 2.2 we covered what machine learning is and the differences between supervised and unsupervised classification. We specifically covered common performance measures and how SVMs work.

- In Section 2.3 we covered software metrics – specifically source code and test suite metrics. We explain how source code metrics can be used to identify code smells and some examples of these metrics. We also explain several approaches to test suite metrics using a combination of source code metrics on test cases as well as coverage metrics.

Chapter 3

Approach

This chapter describes our approach to predicting the mutation score of a source code unit under test based on source code and test suite metrics data. Our approach at a high-level can be summarized in the following steps:

1. Collect mutation score data of the SUT.
2. Collect source code metric data of the SUT.
3. Collect test suite metrics data of the SUT.
4. Synthesize collected data together and store it within a database.
5. Construct classification model.
6. Predict with classification model.

In Section 3.1 we describe each step of our process in detail, according to the overview presented in Figure 3.1. Then, in Section 3.2 we describe how we use the produced classification model of our approach to predict the mutation scores of source code units. We mention related works to our research in Section 3.3.

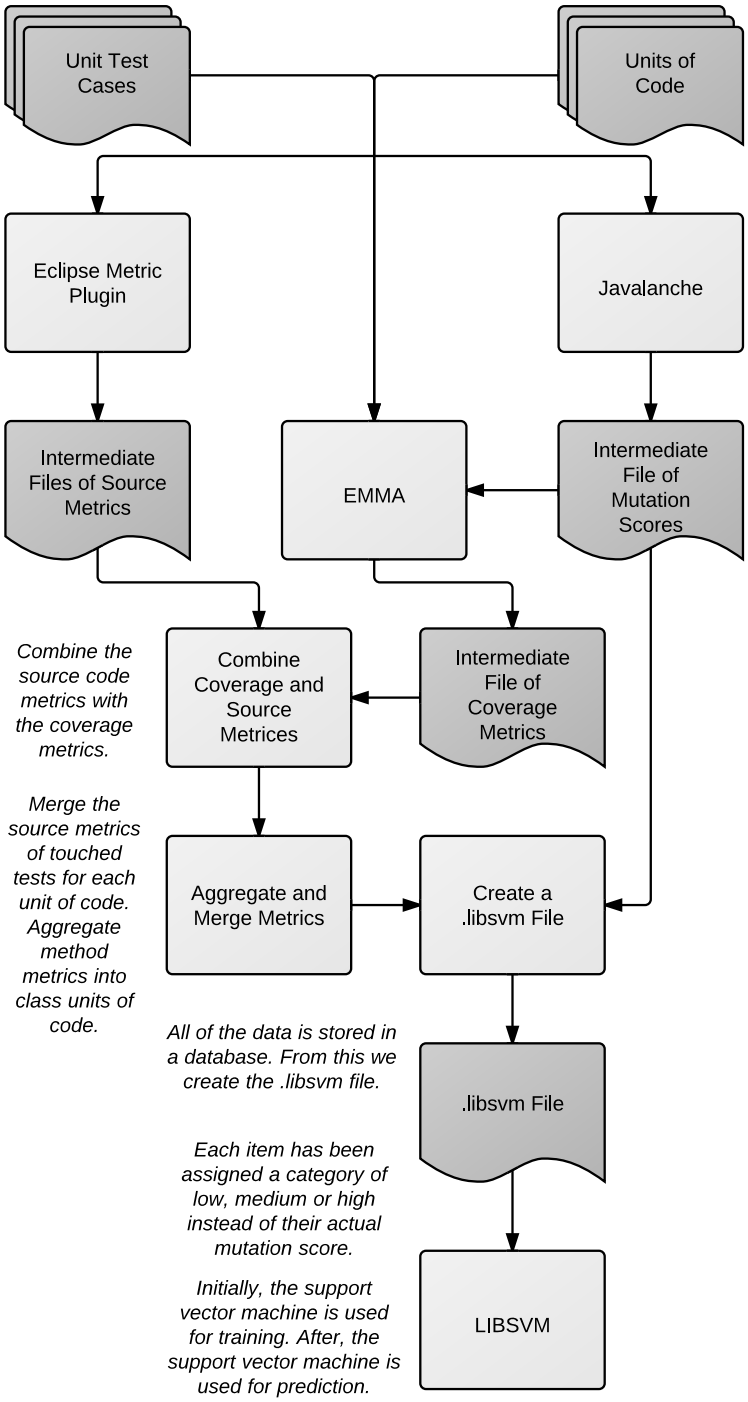


Figure 3.1: Our training process for predicting mutation scores of source code units.

3.1 Process

Our process for mutation score prediction using source code and test suite metrics is shown in Figure 3.1. The complete set of source code and test suite metrics used in our process are shown in Table 3.1. We grouped our metrics into logical feature sets (see Table 3.2) so we could manipulate the groupings later in Chapter 4. This is grouping used to allow to understand the benefits of each feature set with respect to prediction performance. Further mention of the feature sets will be referred to by their corresponding set (from Table 3.2 – ①, ②, ③, ④), and metrics by their abbreviation (from Table 3.1 – *NBD*, *NOF*, *LCOM*, etc. . .). Supervised machine learning techniques require a model first before any predictions are made (as mentioned in Section 2.2). To achieve this, our process first relies on the notion of acquiring known data to construct the appropriate SVM model for future prediction. As mentioned in the motivation (see Section 1.1), our approach aims to reduce the amount of mutation testing done in iterative development. If our technique generalizes well, then it can be possible to build a comprehensive model and predict on different software systems without any prior mutation testing. This will be explored in Chapter 4.

As our approach attempts to predict the mutation score of source code units, we need to keep in mind the factors involved: the source code unit and any unit test cases that provide coverage. Our intuition suggests that we need to look at both source code and test suite metrics to properly measure these two source code artifacts. We reason that since both source code and test suite are tightly coupled for a software system, observing them together would be the best approach for understanding and predicting mutation scores. We hope that by considering the associated unit test cases for a source code unit we can capture a bit on their interactions and relationships in terms of test suite effectiveness.

Metrics	Description	Scope
AMLOC	Average *MLOC* of methods	Class
ANBD	Average *NBD* of methods	Class
APAR	Average *PAR* of methods	Class
ATMLOC	Average *MLOC* of test methods	Class/Method
ATNBD	Average *NBD* of test methods	Class/Method
ATPAR	Average *PAR* of test methods	Class/Method
ATVG	Average *VG* of test methods	Class/Method
AVG	Average *VG* of methods	Class
BCOV	Basic blocks covered in code unit	Class/Method
BTOT	Total basic blocks for code unit	Class/Method
DIT	Depth of inheritance tree	Class
LCOM	Lack of cohesion of methods	Class
MLOC	Method lines of code	Method
NBD	Nested block depth	Method
NOF	Number of attributes	Class
NOM	Number of methods	Class
NORM	Number of overridden methods	Class
NOT	Number of test cases	Class/Method
NSC	Number of children	Class
NSF	Number of static attributes	Class
NSM	Number of static methods	Class
PAR	Number of parameters	Method
SIX	Specialization index	Class
SMLOC	Sum *MLOC* of methods	Class
SNBD	Sum *NBD* of methods	Class
SPAR	Sum *PAR* of methods	Class
STMLOC	Sum *MLOC* of test methods	Class/Method
STNBD	Sum *NBD* of test methods	Class/Method
STPAR	Sum P*AR* of test methods	Class/Method
STVG	Sum *VG* of test methods	Class/Method
SVG	Sum *VG* of methods	Class
VG	McCabe cyclomatic complexity	Method
WMC	Weighted method per class	Class

Table 3.1: The complete set of metrics used as attributes for each vector of the SVM. *The above metrics (listed in alphabetical order) specify the source code unit scope the metric belongs.*

We acquire the source code unit metrics described in Table 3.1 using *Eclipse Metrics Plugin* (further described in Section 3.1.3) and *EMMA* (further described in

Feature Set	Metrics
① – Source Code	*DIT*, *LCOM*, *MLOC*, *NBD*, *NOF*, *NOM*, *NORM*, *NSC*, *NSF*, *NSM*, *PAR*, *SIX*, *VG*, *WMC*
② – Coverage	*BCOV*, *BTOT*, *NOT*
③ – Accumulated Source Code	*AMLOC*, *ANBD*, *APAR*, *AVG*, *SMLOC*, *SNBD*, *SPAR*, *SVG*
④ – Accumulated Test Case	*ATMLOC*, *ATNBD*, *ATPAR*, *ATVG*, *STMLOC*, *STNBD*, *STPAR*, *STVG*

Table 3.2: Feature sets based on a logical grouping (i.e., similar metrics and the means they were acquired) of metrics from Table 3.1.

Section 3.1.4). We aggregate method-level metrics into class-level metrics to follow the scope hierarchy. We also compute the mutation scores using *Javalanche* (further described in Section 3.1.2) and combine those with the source code unit metrics to create our required input for training and prediction.

In Appendix B we investigated the correlation between the mutation score (i.e., what we are predicting) and the individual metrics (i.e., attributes we are using to make the predictions). We found that there approximately six metrics that provided moderate correlation with the mutation score, while the remaining metrics provided only weak or no-correlation.

> *There is no single metric that provides a strong correlation with the mutation score, which suggests this is a difficult prediction to make.*

The following sections walk through the complete process one phase at a time, providing examples where possible. The entire process is executed using our custom scripts that automate data collection, synthesis, and evaluation[1].

[1]Scripts and documentation: `https://github.com/sqrlab/mutation_score_predictor`.

3.1.1 Inputs

To predict the mutation score of class- and method-level source code units, our approach requires: a set of source units of code (the Java files that compose the SUT) and the corresponding set of unit test cases (the JUnit files that compose the test suite for the SUT). A simple example of the required input is presented in Figure 3.2. Our approach is only concerned with source units of code that are being tested, thus the more coverage the test suite provides the more data that can be extracted from the SUT.

3.1.2 Collect Mutation Scores

We use SVM, a supervised learning technique, to predict mutation scores. Before any predictions can occur we must first collect data to compose a feature set with vectors of attributes (i.e., metrics). The collected data must also have their correct categories (i.e., mutation score) assigned to them as we will use the collected data for training purposes. Afterwards, when training is completed, it becomes possible to make predictions on new data based on the model that has been trained.

In our research we use *Javalanche* (version 0.4), a mutation testing tool for Java [SZ09a] that applies a subset of the method-level mutation operators (see Table 3.3). These selected operators provide a close approximation of the effectiveness of using the entire set of method-level operators at a reduced cost [OLR+96].

We chose *Javalanche* for our research because it is customizable and extendable, therefore allowing us to modify *Javalanche* to calculate unit mutation scores and output a richer set of results. Other benefits of *Javalanche* include full integration with JUnit, the use of mutation schemas and bytecode generation to improve performance, and test selection using coverage (see Table 2.3 for full list). Although *Javalanche* does

`Triangle` Source Code

```
public class Triangle {

  public Boolean isValid(int a, int b, int c) {
    if (a <= 0 || b <= 0 || c <= 0)
      return false;
    else if (a + b < c || a + c < b || b + c < a)
      return false;
    return true;
  }

  public TType classify(int a, int b, int c) {
    if (!isValid(a, b, c))
      return INVALID;

    int trian = 0;
    if (a == b)
      trian = trian + 1;
    if (a == c)
      trian = trian + 2;
    if (b == c)
      trian = trian + 3;

    if (trian > 3)
      return EQUILATERAL;
    else if (trian == 0)
      return SCALENE;
    else if (trian == 1 && a + b > c)
      return ISOSCELES;
    else if (trian == 2 && a + c > b)
      return ISOSCELES;
    else if (trian == 3 && b + c > a)
      return ISOSCELES;
    return INVALID;
  }
}
```

`TriangleTest` Test Suite

```
public class TriangleTest {

  public void testScalene() {
    TType type = Triangle.classify(1, 2, 3);
    assertEquals(SCALENE, type);
  }

  public void testIsoceles() {
    TType type = Triangle.classify(2, 2, 3);
    assertEquals(ISOSCELES, type);
  }

  public void testEquiliteral() {
    TType type = Triangle.classify(1, 1, 1);
    assertEquals(EQUILATERAL, type);
  }

  public void testNegative() {
    Boolean isValid = Triangle.isValid(1, -1, 1);
    assertEquals(true, isValid);
  }

  public void testInvalid() {
    Boolean isValid = Triangle.isValid(6, 1, 2);
    assertEquals(true, isValid);
  }

  public void testValid() {
    Boolean isValid = Triangle.isValid(2, 3, 4);
    assertEquals(false, isValid);
  }
}
```

Figure 3.2: Example source code of `Triangle` and its test suite `TriangleTest`.
The above example presents a stripped down example of expected input that our prediction approach requires. This software system is able to classify triangles, and has a few test cases to test its capabilities in classifying triangles.

not have class-level mutation operators, due to the open source nature of *Javalanche* we can extend it to incorporate class-level mutation operators. In addition, *Javalanche* already has concurrency-level mutation operators as well as the ability to identify equivalent mutants using impact analysis.

Using scripts we made our whole approach as automated as possible, thus the user only has to configure a couple variables to target a different software system (i.e., package prefix, test suite name, source directories). We have made minor modifications

Name	Description
REPLACE_CONSTANT	Replace a constant
NEGATE_JUMP	Negate jump condition
ARITHMETIC_REPLACE	Replace arithmetic operator
REMOVE_CALL	Remove method call
REPLACE_VARIABLE	Replace variable reference
ABSOLUTE_VALUE	Insert absolute value of a variable
UNARY_OPERATOR	Insert unary operator

Table 3.3: The set of selective method-level mutation operators used in *Javalanche*.

```
CLASS_NAME,KILLED_MUTANTS,COVERED_MUTANTS,MUTATION_SCORE_OF_COVERED_MUTANTS ...
triangle.Triangle,145,201,0.7213930348258707 ...
```

```
CLASS_NAME,METHOD_NAME,KILLED_MUTANTS,COVERED_MUTANTS,MUTATION_SCORE_OF_COVERED_MUTANTS ...
triangle.Triangle,triangle.Triangle.classify(III)Ltriangle/TriangleType;,84,121,0.6942148760330579 ...
triangle.Triangle,triangle.Triangle.isValid(III)Ljava/lang/Boolean;,61,80,0.7625 ...
```

Figure 3.3: Example CSV files of the mutation scores from the `Triangle` software system.
The above file snippets show the generated class (top) and method (bottom) mutation score CSV files. There are more values related to the number of mutant types generated/killed that are not shown for terseness.

to *Javalanche* that allows it to use all the specified operators from Table 3.3. *Javalanche* is also configured to use its coverage impact analysis to give insight on equivalent mutants (more on this in Chapter 4), though this slows down *Javalanche* substantially. Furthermore we added a custom analyzer that outputs the mutation scores of each class and method units in the SUT.

Javalanche generates all possible mutants, then considers the set of mutants covered by the provided test suite. Given the set of covered mutants, *Javalanche* then tests and records the results of each mutant using its subset of covered test cases for that specific mutant. The newly added analyzer for *Javalanche* then outputs an intermediate CSV file of mutation scores for the covered source code units. Using the example `Triangle` software system presented in Section 3.1.1, the CSV file of the

acquired mutation scores are shown in Figure 3.3. Using the CSV file, we populate a database with all the acquired data, easing the management and analysis of the data.

3.1.3 Collect Source Code Metrics

In our research, we use the *Eclipse Metrics Plugin* (version 1.3.8.20100730-001) to acquire source code metrics of the method- and class-level source code unit under test [Met]. We selected this tool as it provides a comprehensive set of metrics for Java programs (see feature sets ① and ③ from Table 3.2). The metrics can also be exported to XML which is a suitable format from which to extract data. Although this tool is part of Eclipse as a plugin, it is possible to initiate the tool through a CLI interface after importing the SUT into Eclipse. When used the *Eclipse Metrics Plugin* produces an XML file of the source code metrics of the source code units and unit test cases. The produced XML file is *metric-oriented*, so we translate this into a *unit-oriented* format. This phase acquires source code metrics for each source code unit (see feature set ① in Figure 3.2). As we focus on JUnit test cases as our testing framework, we can actually use the *Eclipse Metrics Plugin* to gather the source code metrics of the test suite (see feature set ④ from Table 3.2). Using the example in Figure 3.2 this phase extracts the metrics displayed in Table 3.4 and 3.5.

3.1.4 Collect Test Suite Coverage Metrics

EMMA (version 2.0.5312) is capable of determining the basic block coverage of a test suite [Rou], which is our test suite coverage metrics (see feature set ② in Table 3.2). Using the test suite and the SUT, it is possible to acquire the coverage for each source code unit using the set of covering unit test cases[2]. We run the set of covered unit

[2]Currently we acquire the covered test cases using *Javalanche*, though this can easily be computed solely using *EMMA* with additional analysis.

Table 3.4: Extracted class source code metrics of the `Triangle` software system.

Source Code Unit	*NORM*	*NOF*	*NSC*	*DIT*	*LCOM*	*NSM*	*NOM*	*SIX*	*WMC*	*NSF*
`Triangle`	0	0	0	1	0	0	2	0	20	0
`TriangleTest`	0	0	0	1	0	0	6	0	6	0

Table 3.5: Extracted method source code metrics of the `Triangle` software system.

Source Code Unit	*MLOC*	*NBD*	*VG*	*PAR*
`Triangle.classify`	20	1	13	3
`Triangle.isValid`	5	1	7	3
`TriangleTest.testScalene`	2	1	1	0
`TriangleTest.testIsoceles`	2	1	1	0
`TriangleTest.testEquiliteral`	2	1	1	0
`TriangleTest.testNegative`	2	1	1	0
`TriangleTest.testInvalid`	2	1	1	0
`TriangleTest.testValid`	2	1	1	0

test cases for each source code unit with *EMMA*, producing XML files containing the block coverage of the covered unit test cases on the SUT. We then can extract the coverage of the targeted source code unit. Using the example in Figure 3.2 this phase extracts the following metrics as seen in Table 3.6.

3.1.5 Combine Coverage and Source Metrics

At this point in the process we have acquired all the raw data (mutation scores, source code metrics, and test suite metrics). We can now begin synthesizing data together, combining source code metrics and coverage metrics together. We first analyze all the coverage XML files produced from the coverage phase (see Section 3.1.4). We calculate the coverage that each source code unit has given the set of covered unit test cases used. Now we combine the source code metrics and coverage metrics of a source code unit. The combined data is added to our database to go along with the acquired mutation score. Each source code unit in the database eventually will contain all the metrics pertaining to it, along with its mutation score.

3.1.6 Aggregate and Merge Method-Level Metrics

The last phase for data synthesis is to merge the source code metrics of the covered unit test cases together into their corresponding source code unit. This merger produces feature set ④ from Table 3.2. Using our example, this phase produces the synthesized test suite metrics shown in Table 3.7.

We also aggregate the method-level source code units metrics into their respected parent class-level source code unit. This allows us to populate the database with metrics from feature set ③ from Table 3.2. Using our example, the aggregation of method-level metrics to class-level source code units is shown in Table 3.8.

```
1 1:2 2:1 3:1 4:1 5:0 6:0 7:0.0 8:0.0 9:0.0 10:0.0 11:2 12:2
1 1:2 2:1 3:1 4:1 5:0 6:1 7:3.0 8:1.0 9:1.0 10:0.0 11:2 12:2
2 1:24 2:3 3:1 4:1 5:1 6:1 7:16.0 8:1.0 9:1.0 10:0.0 11:19 12:19
2 1:31 2:6 3:3 4:3 5:1 6:1 7:17.0 8:1.0 9:1.0 10:0.0 11:8 12:18
3 1:1 2:1 3:1 4:1 5:1 6:2 7:16.0 8:1.0 9:1.0 10:0.0 11:3 12:3
3 1:23 2:7 3:2 4:2 5:0 6:0 7:0.0 8:0.0 9:0.0 10:0.0 11:23 12:25
...
```

Figure 3.4: Example file format for *LIBSVM*, a *.libsvm* file of vectors

For a SVM, each row is a vector where the first number in each row is the category and each `<a>:<b>` *represent an attribute. From the above example, there are three categories and 12 attributes. For each attribute* `a` *represents the attribute ID and* `b` *represents the actual value for that attribute. The attribute ID maps to a specific metric that the vector is representing. For example, attribute* `1` *might map to the* `MLOC` *metric, and so-forth.*

3.1.7 Create LIBSVM File

At this point in the process our database contains all the necessary information for the SVM. We have collected and synthesized all the source code and test suite metrics for both class- and method-level source code units. We use *LIBSVM* (version 3.12), a SVM library capable of solving n-group classification problems [CL11]. We decided to use this library implementation as it is mature and used in many other publications[3]. *LIBSVM* has the ability to run entirely from a CLI, and provides an easy to use interface to perform training and prediction. We can now output the specific file format (*.libsvm*) of the acquired data. This format is required for our SVM tool, *LIBSVM*, and contains a list of vectors with each having a category and set of attributes, as seen in Figure 3.4. Our process produces two *.libsvm* files, one for the method-level source code units and another for the class-level source code units.

Instead of predicting a specific mutation score percentage, we categorize all mutation scores as *LOW, MEDIUM, HIGH*, which reduces the mutation score prediction to a three-group classification problem. The ranges of values in each category are determined based on the distribution of the mutation scores in our training data (further explained in Section 4.3.1). Finally, the *.libsvm* file is passed into *LIBSVM*

[3] *LIBSVM* [CL11] has been cited 9323 times according to *Google Scholar* as of May 21st, 2012.

Table 3.6: Extracted coverage test suite metrics (feature set ② of Table 3.2) of the `Triangle` software system.

Source Code Unit	*NOT*	*BCOV*	*BTOT*
`Triangle`	6	60	102
`Triangle.classify`	3	36	72
`Triangle.isValid`	6	24	30

Table 3.7: Merged test suite metrics (feature set ④ in Table 3.2) for each source code unit of the `Triangle` software system.

Source Code Unit	*STMLOC*	*STNBD*	*STVG*	*STPAR*	*ATMLOC*	*ATNBD*	*ATVG*	*ATPAR*
`Triangle`	12	6	6	0	2	1	1	0
`Triangle.classify`	6	3	3	0	2	1	1	0
`Triangle.isValid`	12	6	6	0	2	1	1	0

Table 3.8: Aggregation of method-level source code metrics for each class-level source code unit (feature set ③ of Table 3.2) of the `Triangle` software system.

Source Code Unit	*SMLOC*	*SNBD*	*SVG*	*SPAR*	*AMLOC*	*ANBD*	*AVG*	*APAR*
`Triangle`	25	2	20	6	12.5	1	10	3

to complete the training process. Mutation scores have a range from 0% and 100%, however a SVM cannot perform classification over such a range of real numbers. To circumvent this problem we instead group ranges of mutation scores into groups (i.e., LOW: 0%–33%, MEDIUM: 34%–66%, and HIGH: 67%–100%). As the mutation scores most likely will not follow a balanced distribution we may have to adjust the group ranges to accommodate the distribution. In Section 4.3.1 we examine, the mutation score distribution and consider usable ranges of mutation scores for our categories.

3.2 Prediction

Once we have train the SVM, we can use it for prediction. We can predict the mutation score category of an unknown source code unit by first determining the source code and test suite metrics. The metrics (i.e., attributes) are passed into the SVM model which will then assign a category of *LOW, MEDIUM, HIGH* for the mutation score. As shown in Figure 3.5 our prediction process is not too different from the training process, and we now can exclude *Javalanche* from the process [4].

3.3 Related Work

Although prediction of mutation scores has not been previously researched, the related topic of using software metrics to locate faults in source code has been well researched. For example, Koru and Liu utilized static software measure along with defect data at the class level to predict bugs using machine learning [KL05]. Similarly, Gyimothy et al. used object-oriented metrics with logistic regression and machine learning techniques to identify faulty classes in open source software [GFS05]. Design level metrics were

[4] We currently calculate the *NOT* (i.e., number of tests) metric using *Javalanche*, though ideally we can calculate this using *EMMA*. This is a setback in our current implementation.

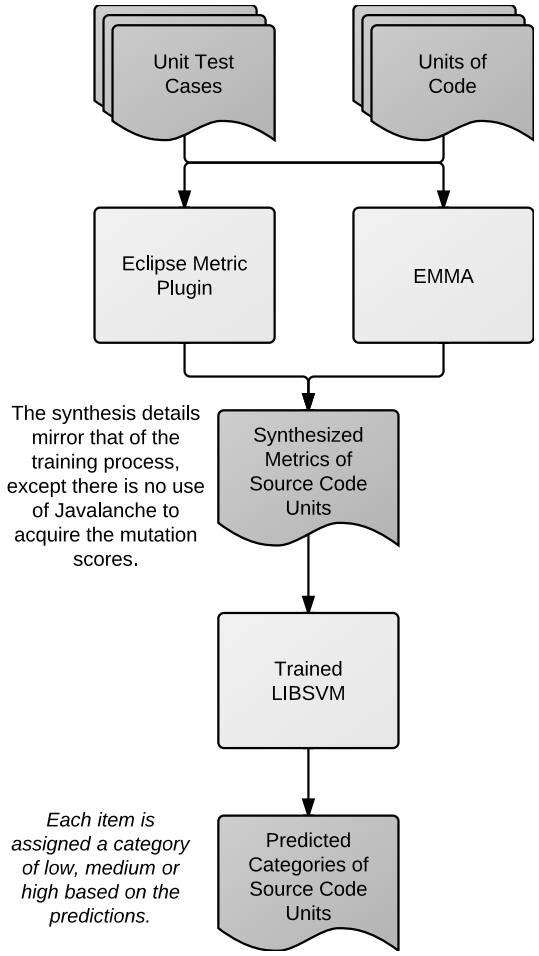


Figure 3.5: Our prediction process for predicting mutation scores of source code units given a trained SVM.

used with a linear prediction model to determine the estimated maintainability and error prone modules of large software systems [MKPS00]. Nagappan et al. used post-release defects of multiple versions along with source code complexity metrics to predict component failures [NBZ06]. Our work is unique in comparison to these previous works since we not only use source code metrics but we also use test suite metrics to enhance our predication capabilities. The aforementioned works on fault prediction do not use test suite metrics for their predictions, however they do, in some cases, utilize bug reports.

Wong et al. used statement coverage of test cases to localize faults using different heuristics [WDC10]. The aforementioned studies primarily utilized source code metrics, however this study used only test suite metrics for fault localization. On a more related study, Anderson et al. used a neural network to prune a test suite, which preserved test suite effectiveness for domain based testing [AMM95]. Their approach used attributes of test cases as input and an oracle that determined the severity of faults present in test cases. The study Anderson et al. conducted did not examined the SUT's source code metrics, and only focused on the test suite. They used test suite metrics such as the length of the test case and command/parameter frequency.

Nagappan et al. created the *Software Testing and Reliability Early Warning (STREW) metric suite*, a test quality indicator [NWO+05,NWVO05]. Both source source code and test suite metrics were used in their calculation of test quality, which was the closest use of metrics to our own set.

Inozemtseva researched the relationship between test suite size, basic block coverage and test suite effectiveness [Ino12]. This study used *EMMA* to measure basic block coverage and *Javalanche* to measure test suite effectiveness, which is very similar to our approach. The research question between their study and ours is quite different. We are trying to predict the mutation score (i.e., the test suite effectiveness) of individual source code units. Inozemtseva's study attempted to understand whether basic block coverage and test suite size are effective in predicting test suite effectiveness. Inozemtseva determined that basic block coverage is a poor predictor of test suite effectiveness.

3.4 Summary

In this chapter we covered all aspects of our approach in terms of tools and the process used to collect and train our SVM. As we use a SVM as our prediction technique we require training data, thus we collected the mutation scores of each source code unit. We also collected the various metrics from the source code and test suite and synthesized all the acquire data to train our SVM to make prediction on existing and new data. We described each step of our process in Section 3.1, and how we perform prediction in Section 3.2. Finally in Section 3.3 we addressed related work to our approach on prediction of mutation scores using machine learning prediction techniques.

Chapter 4

Empirical Evaluation

In this chapter, we evaluate our approach detailed in Chapter 3. We describe how we setup and conduct our empirical evaluation in Section 4.1 and we describe our experimental method in Section 4.2. Finally we discuss our experimental results in Section 4.3 and threats to validity in Section 4.4.

4.1 Experimental Setup

To encourage reproducibility of our empirical evaluation we discuss the specific details concerning environment (Section 4.1.4), tool configuration (Section 4.1.1), test subjects (Section 4.1.2), and data preprocessing (Section 4.1.3).

4.1.1 Tool Configuration

We use three tools in our approach – *Javalanche*, *Eclipse Metrics Plugin* and *EMMA*. For all three tools, our approach manipulates the raw output of these tools to better support data synthesis. We use the default configuration for *Eclipse Metrics Plugin* and *EMMA*, as these are already configured to provide the necessary data.

We configured *Javalanche* to better suite our approach. Although *Javalanche* has the ability to run parallel tasks, we did not utilize this feature. This was to avoid unnecessary issues that can occur due to concurrent access to file resources that a test suite may use. We enabled the coverage impact analysis of *Javalanche* as it provides comprehensive data regarding the mutants such as the type, location, and whether or not it was killed. The additional analysis is useful and required in the implementation of our approach, though it reduces the performance of *Javalanche*.

We perform 10-fold cross-validation as described in Section 2.2. We used the default values for most *LIBSVM* parameters including using the default kernel function (the RBF kernel as it comes recommended by the authors [HCL03]). Although we use the default kernel function (RBF) we do vary the parameters *gamma* and *cost*. *LIBSVM* assists in the selection of the kernel function parameters by providing a *script* that automatically scales the data and selects the kernel parameters using a grid search [HCL03]. A grid search iterates over a range of parameters (i.e., gamma and cost for the RBF kernel) while measuring the effect it has on the classifiers performance. Parameter selection is a critical aspect of machine learning algorithms, and it can greatly influence the classification accuracy. We allow *LIBSVM* to automatically take care of this to best select the kernel parameters based on the provided data. We allow *LIBSVM* to use eight threads for computation tasks.

4.1.2 Test Subjects

We constructed three simple criteria to select our test subjects:

- We selected software systems that have a minimum of 5000 total SLOC. We decided to use this size as our minimum to avoid selecting *toy* software systems

that are not similar to *real* software systems. Also, by using *real* software systems we can potentially collect more data than that of *toy* software systems.

- We selected open source projects as they are relatively easy to acquire as opposed to industry projects, and are freely available to analyze.

- Our approach required projects with a test suite or set of test cases as it is fundamentally required for mutation testing.

Following the criteria outlined, we selected the following eight open source Java software systems shown in Table 4.1. A brief description of each open source software system used in our empirical evaluation is presented as follows:

- ***logback-core***: *"Logback is intended as a successor to the popular log4j project, picking up where log4j leaves off. The logback-core module lays the groundwork for the other two modules"* [log].

- ***barbecue***: *"Barbecue is an open source, Java library that provides the means to create barcodes for printing and display in Java applications"* [bar].

- ***jgap***: *"JGAP is a Genetic Algorithms and Genetic Programming component provided as a Java framework"* [jga].

- ***commons-lang***: *"The standard Java libraries fail to provide enough methods for manipulation of its core classes. Apache Commons Lang provides these extra methods"* [com].

- ***joda-time***: *"Joda-Time provides a quality replacement for the Java date and time classes. The design allows for multiple calendar systems, while still providing a simple API"* [jodb].

Table 4.1: The set of test subjects along with source code and test suite metrics.

Test Subject	**Source LOC**	**Source Classes**	**Source Methods**	**Test LOC**	**Test Classes**	**Test Methods**	**Test Cases**
logback-core (1.0.3) [log]	12118	249	1270	8377	174	688	286
barbecue (1.5-beta1) [bar]	4790	58	299	2910	38	416	225
jgap (3.6.1) [jga]	28975	415	3017	19694	180	1633	1355
commons-lang (3.1) [com]	19499	149	1196	33332	242	2408	2050
joda-time (2.0) [jodb]	27139	227	3635	51388	221	4755	3866
openfast (1.1.0) [ope]	11646	265	1447	5587	115	421	322
jsoup (1.6.2) [jso]	10949	198	954	2883	25	335	319
joda-primitives (1.0) [joda]	11157	128	1868	6989	49	746	1810
all	**126273**	**1689**	**13686**	**131160**	**1044**	**11402**	**10233**

- ***openfast***: *"OpenFAST is a 100% Java implementation of the FAST Protocol (FIX Adapted for STreaming). The FAST protocol is used to optimize communications in the electronic exchange of financial data"* [ope].

- ***jsoup***: *"jsoup is a Java library for working with real-world HTML. It provides a very convenient API for extracting and manipulating data, using the best of DOM, CSS, and jquery-like methods"* [jso].

- ***joda-primitives***: *"Joda Primitives provides collections and utilities to bridge the gap between objects and primitive types in Java"* [joda].

For our experiments we have eight test subjects that we can use individually, as well as consider them collectively. Therefore, we further refer to the collective set of all the individual test subjects as the *all* subject. By combining the individual test subjects we can evaluate our approach on a mixed set of data. Though each test subject is unique in terms of the functionality they provide and the specific structural design choices, each one shares the commonality of being a software system. To further explore how our approach performs on different data, we consider eight additional sets using the *all* subject as the base, excluding an individual test subject. In other words, we have eight *all_ but_ <subject>* subjects, which is a combination of each individual test subject except the *<subject>*. These additional subjects allow us to evaluate our prediction approach by keeping one test subject completely isolated from the rest. As our approach focuses on prediction of mutation scores using metrics of the test subjects, the *all* and *all_ but_ <subject>* subjects allow us to evaluate the generalizability of our approach on mixed and isolated test subjects.

4.1.3 Data Preprocessing

We first run each test subject through a verification *test* that *Javalanche* provides. This test identifies any unit test cases that cannot execute properly or fail within *Javalanche*. We have to remove these test cases as the mutation testing process requires a test suite with no errors. We then import all the test subjects into Eclipse, as that is where the *Eclipse Metrics Plugin* operates. With our approach, we simply configure our scripts to identify the test subject to collect data from, and the results are then stored in a database.

4.1.4 Environment

We conducted all of the experiments for our empirical evaluation on a single machine that has six GB of random access memory, a hard drive disk running at 7200 revolutions per minute and an Intel Core i7-870 processor running at 2.93 gigahertz. The environment is relevant as the mutation testing performance cost is dependant on the processor and hard drive disk speed.

4.2 Experimental Method

Our empirical evaluation consists of five separate experiments:

- **Mutation Score Distribution (Section 4.3.1)**: Using our test subjects described in Section 4.1.2, we first want to understand their mutation testing results. Using our approach, we collect the source code, test suite metrics, and mutation scores of each test subject. As each test subject consists of multiple class- and method-level source code units, we are interested in the distribution of mutation scores. By understanding the distribution, we can gauge the available

data that we will have for the later experiments and detect possible anomalies. We can also identify classification categories for the future experiments based on the mutation distribution as SVM have difficulty predicting real values (i.e., the mutation score).

- **Cross-Validation (Section 4.3.2)**: We identified features that describe source code units in Table 3.1. We grouped the features into sets (see Table 3.2) so we could evaluate how each set influences our classification performance. In this section we acquire the cross-validation accuracy for using the different feature sets over all the available data to identify the best feature set. We then evaluate the cross-validation accuracy over each individual test subject using the best set of features.

- **Prediction on Unknown Data (Section 4.3.3)**: Cross-validation accuracy provides a good indicator of classification performance, however it does not simulate realistic prediction on unknown data. In this section, we control the training and testing data for our SVM to evaluate the prediction accuracy on unknown data. Using the *all_but_*<*subject*> subjects, we can evaluate the prediction accuracy when dealing with a unknown test subject that is isolated from the training data. We also evaluate the prediction accuracy of unknown data within an individual test subject. Both of these predictions are on unknown data but they explore the performance differences for prediction within a test subject, and against a different test subject.

- **Optimization and Generalization (Section 4.3.4)**: It is not known prior to predicting on unknown data what parameter values to use for the SVM. In Section 4.3.3, the parameters are selected based on what maximizes the cross-validation accuracy with hopes that these parameters generalizes to the

unknown data. In this section, we identify the most appropriate parameters that generalize to unknown data prediction. We evaluate the implications of using the generalizable parameters with respect to their impact on predicting unknown data.

- **Impact of Training Data Availability on Prediction Accuracy (Section 4.3.5)**: Using the findings from the previous experiments, we explore the impact of data availability on prediction accuracy. We graph the prediction accuracies against the amount of training data used for prediction. This experiment evaluates the applicability of using our approach in iterative development where it is beneficial to avoid evaluating every mutant generated.

4.3 Experimental Results

The following five sections present experiments used in our empirical evaluation for our approach. Each section starts with a general research question that is addressed throughout the section.

4.3.1 Mutation Score Distribution

Research Question #1: *What is the mutation score distribution of our test subjects?*

Research Question #2: *Using the distribution of our test subjects' mutation scores can we identify three categories of mutation scores to predict?*

As described in Chapter 3, our approach uses mutation testing to acquire the necessary mutation score data used for the category of the source code units. Table 4.2 shows

Table 4.2: Mutation testing results of the test subjects from Table 4.1.

Test Subject	Mutants Generated	Mutants Covered	Coverage Percent (%)	Mutants Killed	Mutation Score (%)[a]	Time Taken (*hh:mm:ss*)
logback-core	10682	7350	68.8	5400	73.5	01:49:10
barbecue	27324	4339	15.9	2727	62.8	00:49:51
jgap	31929	17903	56.1	13328	74.4	07:04:44
commons-lang	45141	41761	92.5	33772	80.9	15:51:59
joda-time	70594	58595	83.0	48545	82.8	31:55:50
openfast	14910	8371	56.1	6869	82.1	01:34:38
jsoup	14165	10540	74.4	8430	80.0	03:55:56
joda-primitives	22269	17334	77.8	13499	77.9	01:24:33
all	**237014**	**166193**	**70.1**	**132570**	**79.8**	**64:26:41**

[a] Mutation score is calculated using the covered and killed mutants.

Table 4.3: The usable number of source code unit data points gathered from the test subjects in Table 4.1.

Test Subject	Class-Level	Method-Level
logback-core	115	447
barbecue	31	143
jgap	124	655
commons-lang	124	789
joda-time	194	2019
openfast	120	401
jsoup	83	381
joda-primitives	73	675
all	**864**	**5510**

the results of running *Javalanche* on our test subjects. For all of our test subjects, mutation testing produced a large number of mutants that were evaluated, taking approximately 64 hours to complete in our experimental environment. As described in Section 2.1.2, *Javalanche* utilizes *coverage* (i.e., basic block coverage) for test selection which limits the number of mutants to be evaluated to a subset of covered mutants. *Javalanche* was able to kill 79.8% of the covered mutants using all projects cumulatively. All individual projects, except for *barbecue*, exceeded a mutation score of at least 73% which indicates that the test suites for covered source code units are reasonably effective. The overall coverage is 70.1%, indicating that the test suites of the projects did not cover the remaining 29.9% of the generated mutants. Realistically, mutation scores are calculated using the entire project's source code, but for our purpose only covered mutants were used. The corrective action for non-covered mutants (i.e., mutants not covered by test suite using basic block coverage) is to add new test cases that provide coverage over the mutant's location.

Source code and test code metrics were collected as described in Chapter 3, which represent the set of feature data that make up the vectors of our SVM. Our approach can only make predictions using the synthesis of both mutation score data (i.e., category data) and source and test suite metrics (i.e., feature data) of source code units. If any piece of data is missing, we cannot use that source code unit for training and prediction purposes. Using our approach, we collected data for 864 class-level and 5510 method-level source code units (see Table 4.3). We ignored abstract, anonymous, and overloaded source code units, because taking these into account would be a complex task. In addition to the ignored source code units we also ignored units with missing data (i.e., no tests cases), restricting the available data for further experiments. We present the distribution of the all the collected data points[1] for

[1] *Data points* are the *vectors* or *rows of data* within a SVM *.libsvm* file.

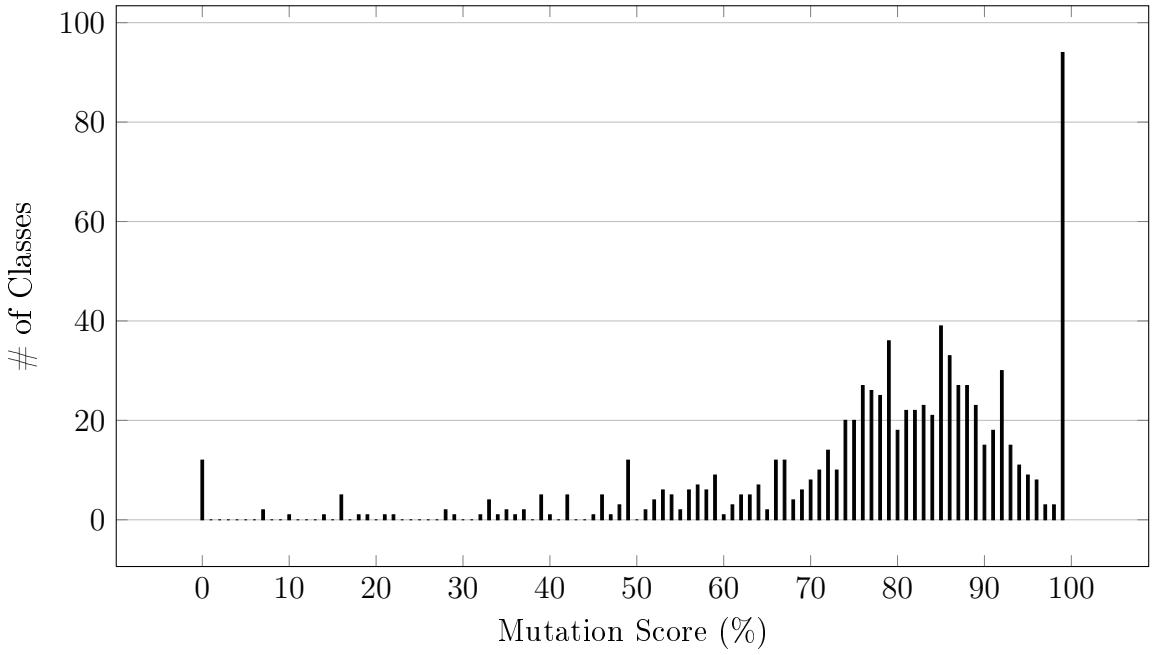


Figure 4.1: Mutation score distribution of classes from all eight test subjects from Table 4.1 that can be used for training.

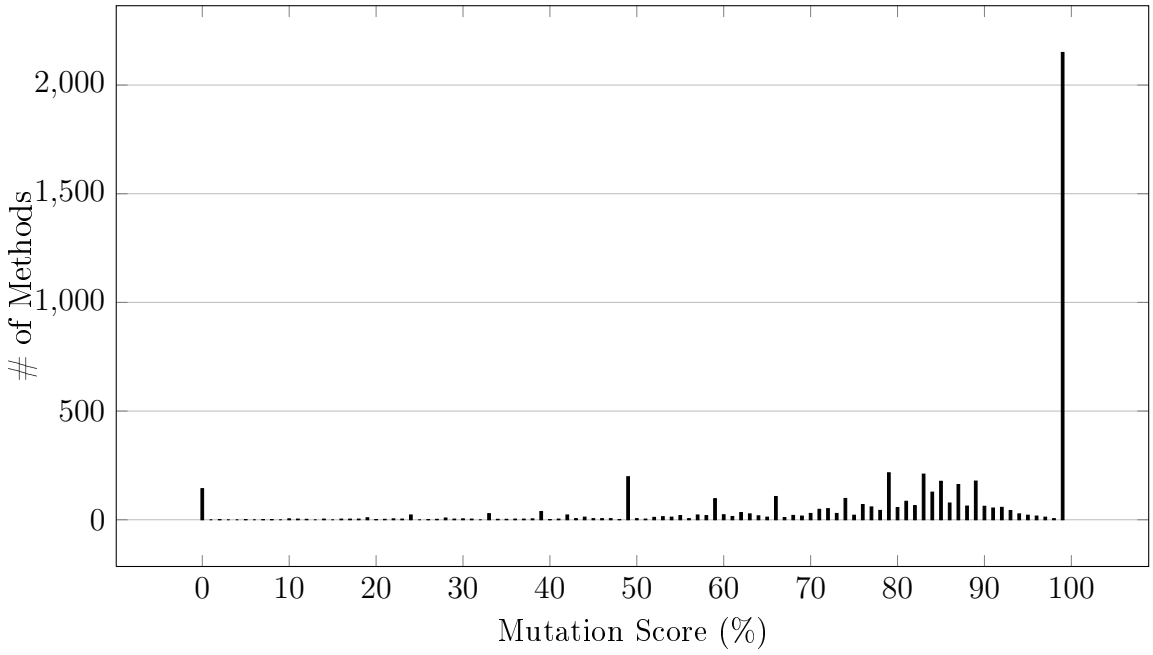


Figure 4.2: Mutation score distribution of methods from all eight test subjects from Table 4.1 that can be used for training.

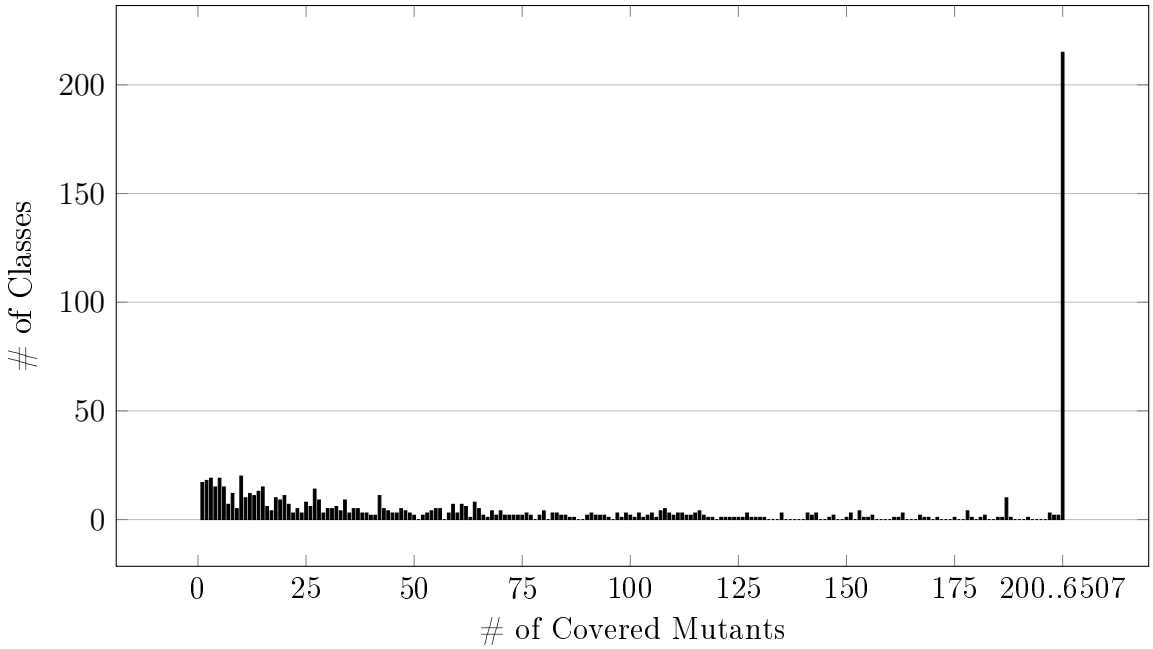


Figure 4.3: Covered mutant distribution of classes from all eight test subjects from Table 4.1 that can be used for training.

The above figure presents a subset of the full distribution by collapsing the values that exceed 200 into a single data point. The max number of covered mutants found was 6507, which corresponds to the following class `org.joda.time.format.ISODateTimeFormat` *from the* `joda-time` *project.*

both class-level and method-level source code units with respect to mutation score in Figure 4.1 and 4.2. The mutation score distributions for each individual project are found in Appendix A. The mutation distribution of both class- and method-level source code units are both negatively skewed, confirming our earlier observation that the test suite for the collected source code units are reasonably strong at detecting faults. We noticed that there were spikes comparative to their surroundings at the 0%, 50% and 100% mutation score values, in particular the 100% value is two to nine times greater then other areas respectively. We speculate the spikes occur because a large number of source code units probably have small number of covered mutants (i.e., easier to kill all, evenly kill half or kill none). Analysis of the covered mutant distribution for class-level source code units (as seen in Figure 4.4) shows a slightly

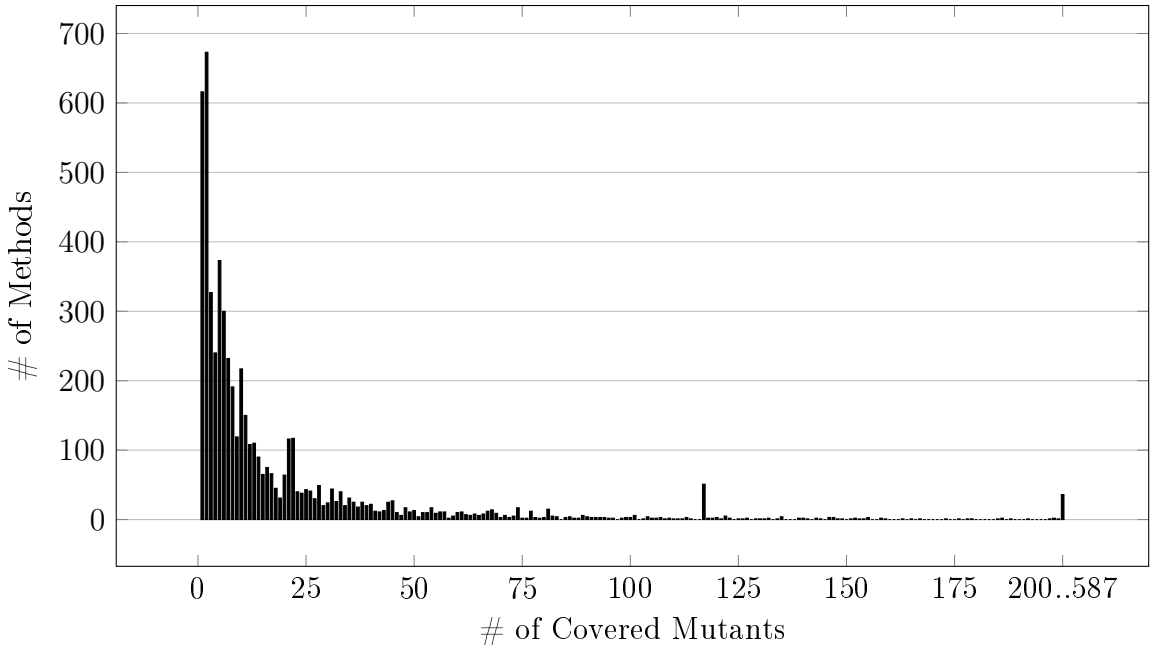


Figure 4.4: Covered mutant distribution of methods from all eight test subjects from Table 4.1 that can be used for training.

The above figure presents a subset of the full distribution by collapsing the values that exceed 200 into a single data point. There were 51 methods that had 117 covered mutants each, of these 48 are similar and are contained within the `ISODateTimeFormat` *class of* `joda-time`*. The max number of covered mutants was 587, which corresponds to the* `org.joda.time.format.PeriodFormatterBuilder$FieldFormatter.parseInto` *method.*

denser grouping for low covered mutants. The positively skewed distribution of covered mutants for method-level source code units supports our speculation. With respect to the percentile of the class-level distribution of covered mutants, a quarter of the classes have less then 16 covered mutants. The method-level results show that half of the method have less then six covered mutants, and a quarter of the methods have less then two covered mutants. The distributions of covered mutants confirm our speculation that many of the source code units have a low number of covered mutants, which can contribute to the high 0%, 50% and 100% mutation scores.

In Section 3.1.7, we mentioned that our approached would create *.libsvm* files using the acquire data. The category data required for the files that the SVM utilized based

on the mutation score of source code units. To avoid predicting the exact mutation score (i.e., a set of real numbers), we instead used an abstracted set of categories (i.e., ranges of mutation scores). We were unsure how to properly select the ranges to use for our categories, but eventually decided to base our categories on the distribution of mutation scores from Figure 4.3 and 4.4. We found that the class-level distribution has a 25^{th}, 50^{th} and 75^{th} percentile of 72%, 81%, 89% respectively, and for same percentiles of the method-level distribution are 75%, 87%, 99%. Using these values we decided to use the following as our general case of categories for all further experiments:

- **LOW** = [0% – 70%)
- **MEDIUM** = [70% – 90%)
- **HIGH** = [90% – 100%]

The rational behind the categories is to separate the lower and upper percentiles in to the LOW and HIGH category, with the remaining into the MEDIUM category. We believe that these values will provide a sufficient level of information over the mutation testing coverage of the source code units.

4.3.2 Cross-Validation

> ***Research Question:*** *Using the test suite and source code data from our test subjects can we identify a set of features that maximize cross-validation accuracy?*

Using the determined classification categories from the previous section, we have imbalanced training data for class- and method-level source code units as shown in Table 4.4. Imbalanced data occurs when the data is not evenly distributed across the classification categories. This can be problematic for supervised machine learning

Category	Mutation Score Range	Class-Level	Method-Level
LOW	[0% – 70%)	191	1104
MEDIUM	[70% – 90%)	459	1782
HIGH	[90% – 100%]	214	2624

Table 4.4: The available number of source code units that fall within the determined ranges of mutation scores.

techniques as they will heavily classify towards the majority category [BOSB10]. Barandela et al. indicate that there are three strategies to reduce the problem of imbalanced training data: Over-sample, under-sample, or internally bias the classification process [BVSF04]. It was shown that simple random under-sampling can be an effective solution (though not always the best) to this problem [Jap00, AKJ04]. As Akbani et al. mentioned *"... we are throwing away valid instances, which contain valuable information"* [AKJ04], therefore we might be limiting the ability to generalize to new unknown data. The alternative is to perform over-sampling, as Batista et al. mention *"... random over-sampling can increase the likelihood of occurring overfitting, since it makes exact copies of the minority class examples"* [BPM04]. We decided to utilize random under-sampling as it provides a simple approach to dealing with imbalanced data. Furthermore, the imbalanced data is not too severe (minority to majority ratio is approximately two to five), thus we believe we are not losing that much information by under-sampling our training data.

To evaluate the cross-validation accuracy of the acquired data we randomly under-sample the data to balance the amount of data points within each category. We utilize 191 class-level and 1104 method-level source code units data points from each category, these values are chosen to maximize the number of data points from the minority category using random under-sampling. We use the *LIBSVM* [CL11] library to perform 10-fold cross-validation over the undersampled data.

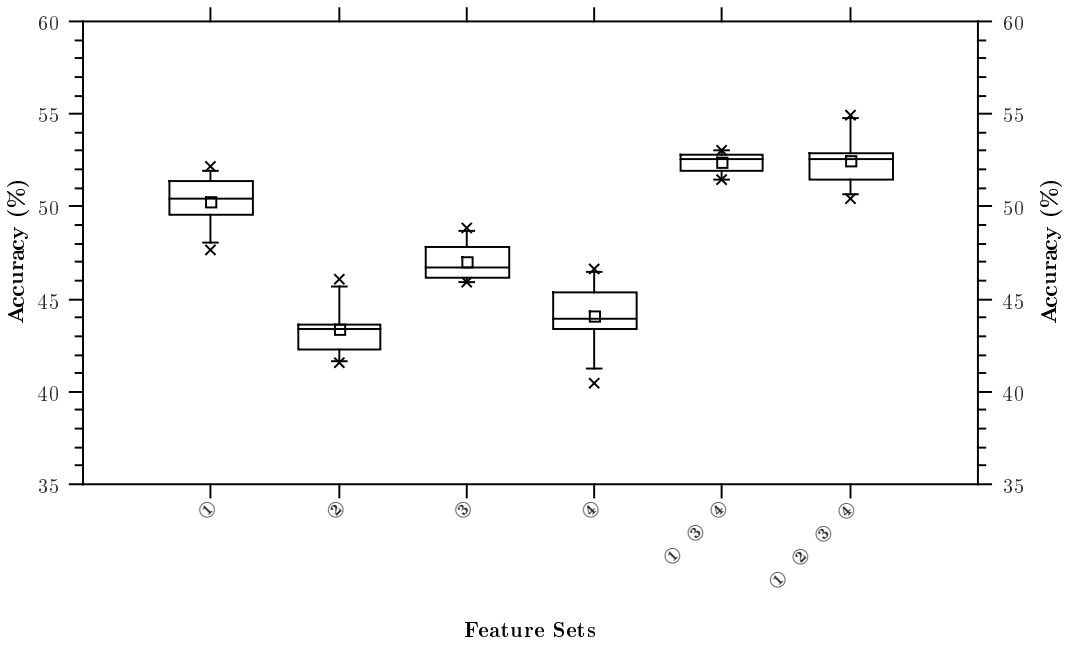


Figure 4.5: Class-level cross-validation accuracy of feature sets on the *all* subject.

All further box plots can be interpreted as follows: The large box for represents the 25^{th}–75^{th} percentile range. Within this large box there is a smaller white square which represents the mean value, while the horizontal line represents 50^{th} percentile (i.e., median). Extending from the large box on either side are the whiskers which extend to the 5^{th} and 95^{th} percentile in either direction. Past the end of the whiskers are small diagonal crosses (i.e., X) which represent the min and max values. Recall that random with respect to our approach is 33.3% due to undersampling to three categories, which is later shown as a dotted line between the y-axis of the box plot.

Recall that we have a set of features (i.e., attributes for our vector in the SVM) as listed in Table 3.2. We explore the cross-validation accuracy using different feature sets (i.e., ①, ②, ③, ④) in Figures 4.5 and 4.6. The cross-validation accuracy of class- and method-level source code units is performed on the collective *all* subject over ten executions to account for random undersampling of our data. To assess our cross-validation accuracy, we use random selection as our baseline. In our case, since we undersample our three categories, random selection will have an accuracy of 33.3%. We can see that all feature sets are able to outperform random which indicates that

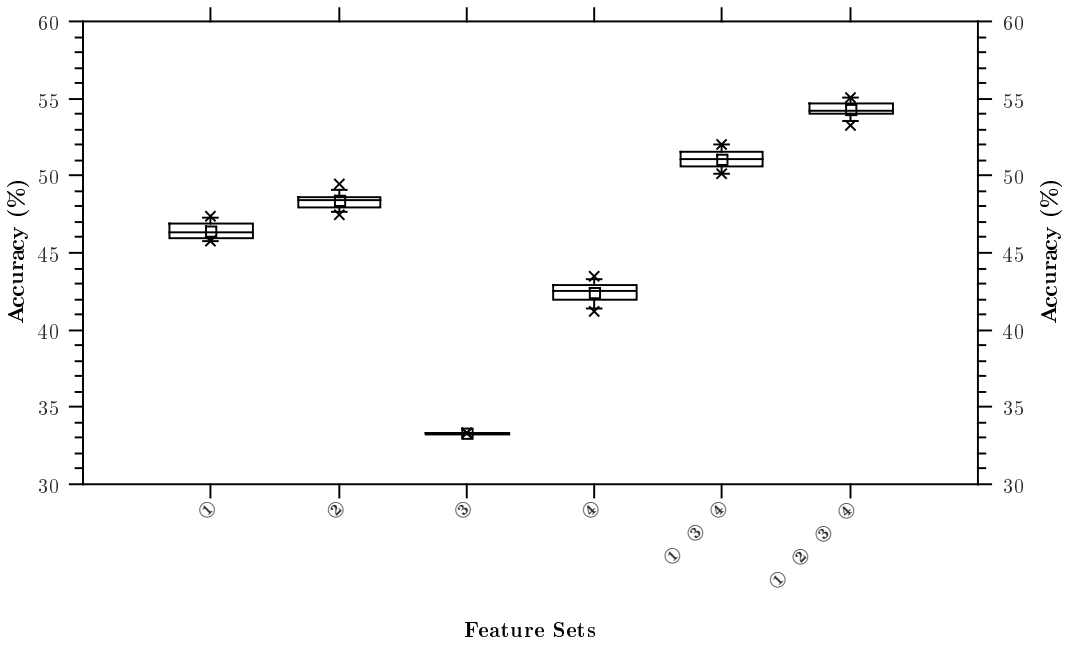


Figure 4.6: Method-level cross-validation accuracy of feature sets on the *all* subject.

there is some predictive power in the selected feature sets[2]. We include a subset of all features (feature sets ① ③ ④) to show the effects of merging only the source code and test suite metrics (excluding coverage feature set ②). We can clearly see that by using all feature sets together we can acquire higher cross-validation accuracy then using the feature sets individually. This observation supports our intuition that using various source code and test suite metrics together can predict the mutation scores of source code units well.

We have looked at the overall cross-validation accuracy using the different feature sets and found that using all feature sets provides the best accuracy. To understand how different projects behave when we apply our technique, we consider each test subject independently using all the feature sets. Figure 4.7 illustrates the class-level cross-validations of each test subjects with the *all* subject as a comparison. We can

[2]Feature set ③ for method-level source code units (Figure 4.6) does not add any value (as it is specifically tailor for class-level source code units), and thus performs at random.

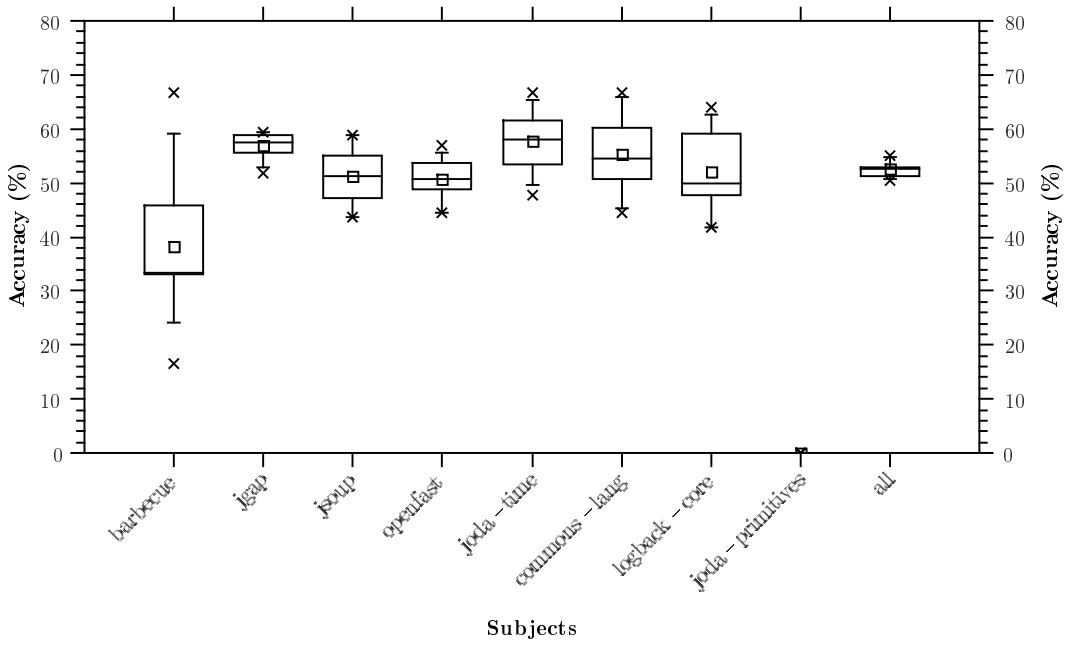


Figure 4.7: Class-level cross-validation accuracy of each test subject using all feature sets (① ② ③ ④).

Test Subject	Class-Level	Method-Level
logback-core	12	138
barbecue	2	36
jgap	27	197
commons-lang	18	132
joda-time	21	259
openfast	24	73
jsoup	13	58
joda-primitives	1	165
all	191	1104

Table 4.5: The number of data points used for each category based on undersampling the lowest category to provide balanced data, for each test subject.

see that all but *barbecue* and *joda-primitives* are similar with respect to mean accuracy. All of the independent test subjects have larger variation in their accuracies compared to the method-level accuracies, which is most likely due to the limited data that each test subject provides on its own. Recall that we are undersampling our data to

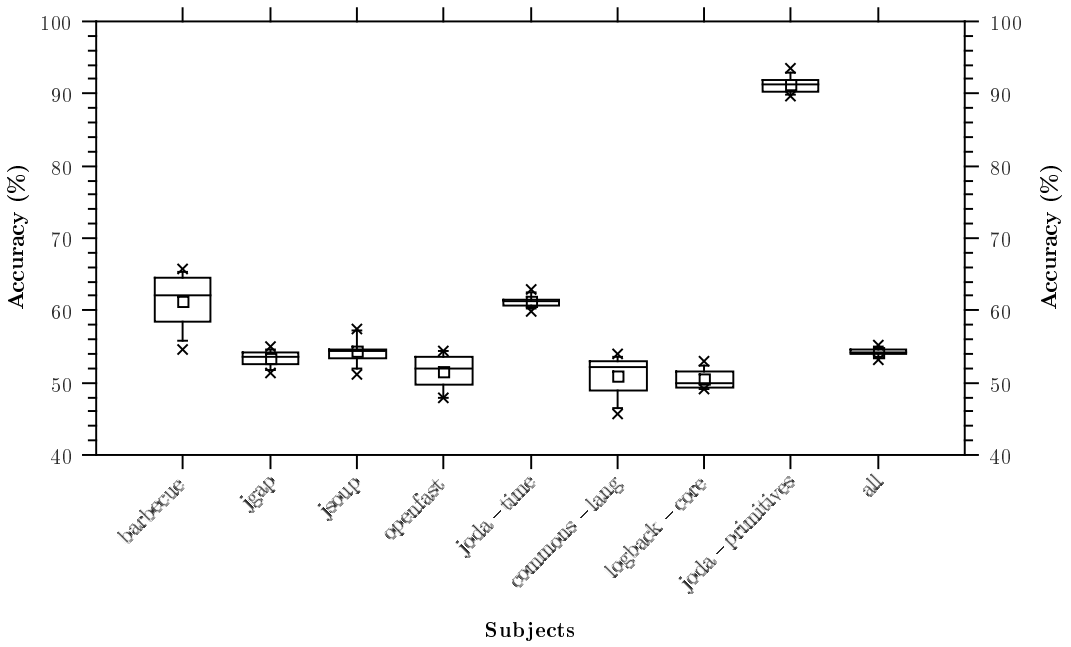


Figure 4.8: Method-level cross-validation accuracy of each test subject using all feature sets (① ② ③ ④).

achieve balanced categories, thus in some situations the amount of data being used can drastically be reduced. In the case of *barbecue*, the undersampling only allowed two instances of data to be used for each category, which explains the huge variation that it has. *joda-primitives* has only one instance for each category, which resulted in a cross-validation accuracy of 0%. Table 4.5 provides details regarding the number of data points being used with undersampling. Moving on to the method-level source code units presented in Figure 4.8, we can see that all but *barbecue*, *joda-primitives*, and *joda-time* are comparable to the *all* accuracy with slightly larger variations. Again, *barbecue* has a low number of data points being used which can explain the larger variations in accuracy. With *joda-primitives*, we have an unusually high cross-validation accuracy. If we look at Figure A.8 in Appendix A, we can see that the mutation score distribution is cleanly separated according to the category ranges (i.e., a large number of 100% and 50% mutation score methods, with the remaining between these values).

It might just be the case that the *joda-primitives*'s data is easier to separate with the SVM, thus allowing it achieve higher accuracy. *joda-time* presents a slightly higher cross-validation accuracy then the other test subjects. This could be because it has the most data points available for the SVM or because it has 48 methods that are very similar (most likely duplicates) as we saw in the covered mutant distribution (see Figure 4.4). Similar methods would be classified in the same category, thus this could slightly inflate the cross-validation accuracy if this was the case.

4.3.3 Prediction on Unknown Data

> ***Research Question #1:*** *How well can our approach predict on unknown data, within an individual software system?*
>
> ***Research Question #2:*** *How well can our approach predict on unknown data, accross software systems?*

By using *LIBSVM* we want to train a classifier such that it can predict well on *unknown data*. In our experiment we consider unused data from undersampling to be unknown as it is not used during training. Parameter selection (i.e., for gamma and cost of a RBF kernel) and the training/testing data sets play an important role in developing a good classifier. Ultimately, we want to obtain a classifier that is able to *generalize* to new, unknown data. In our specific case, we have eight different test subjects (that are most likely not similar to each other in terms of features) in which we want to maximize our performance at predicting unknown data. We used cross-validation in Section 4.3.2 as it mitigates the overfitting problem introduced by training (i.e., a model becomes specifically tuned for the training data set) [HCL03]. Parameter selection also occurred automatically using a grid search approach to find

Test Subject	Class-Level [LOW/MEDIUM/HIGH]	Method-Level [LOW/MEDIUM/HIGH]
logback-core	36/43/0	0/3/30
barbecue	13/12/0	20/15/0
jgap	24/19/0	26/0/38
commons-lang	0/65/5	0/186/207
joda-time	0/87/44	0/296/946
openfast	0/33/15	0/69/113
jsoup	0/18/26	0/50/157
joda-primitives	0/64/6	0/105/75
all_but_logback-core	48/55/12	138/141/168
all_but_barbecue	15/14/2	56/51/36
all_but_jgap	51/46/27	223/197/235
all_but_commons-lang	18/83/23	132/318/339
all_but_joda-time	21/108/65	256/555/1205
all_but_openfast	24/57/39	73/142/186
all_but_jsoup	13/31/39	58/108/215
all_but_joda-primitives	1/65/7	165/270/240

Table 4.6: The number of data points present in each category for each test subject's prediction data set after undersampling (if possible) has occurred.

a set of parameters that maximized the cross-validation accuracy on the training data set.

We conducted a number of tests where we use *LIBSVM*'s *easy script* to find the best parameters that maximize cross-validation accuracy and then apply the classifier to unknown data. We continue to undersample our data and conduct each experiment ten times to determine the prediction accuracy. We are interested in the prediction accuracy of unknown data within a system as well as across systems. See Table 4.6 for the number of unknown data items being used for each subject with respect to predictions on unknown data. For within a system we train on the undersampled data of an individual test subject and predict on the remaining unknown data (i.e., what is left from the undersampling). For across systems we can consider the *all_but_* <*subject*> subjects, where we train our classifier on all but the <*subject*>

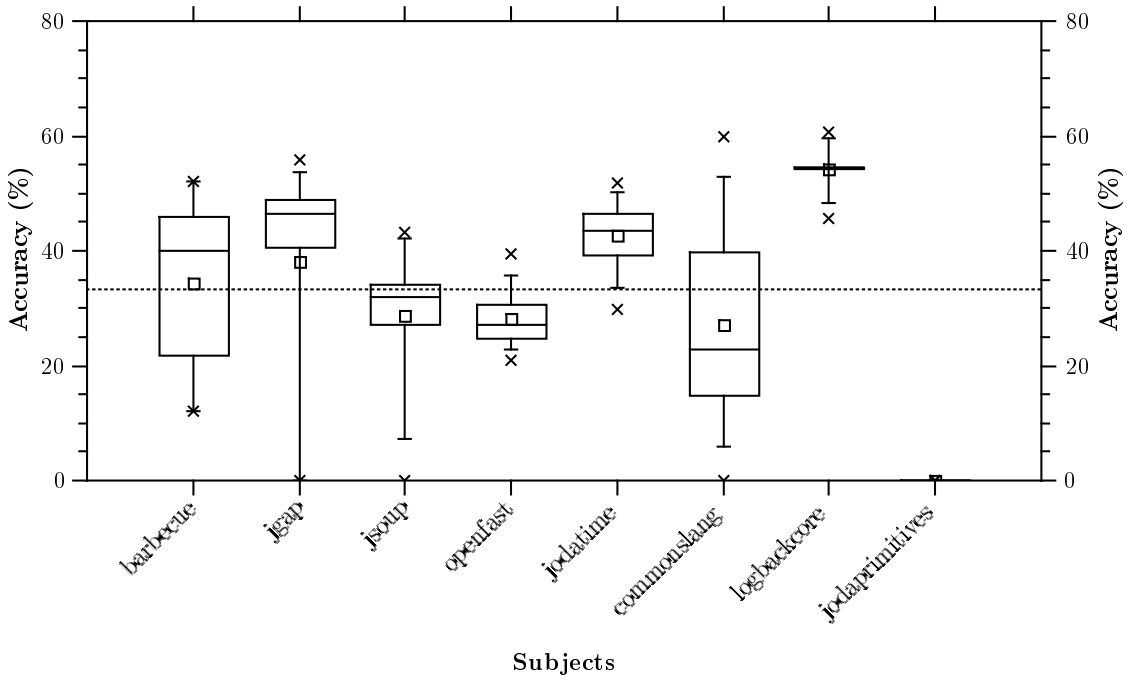


Figure 4.9: Class-level training and prediction accuracy on unknown data within a system.

---

and then predict on the excluded test subject. Section 4.3.3.1 presents results for prediction within a system, and Section 4.3.3.2 presents results for prediction across systems.

4.3.3.1 Prediction Within a System

This experiment explores the capability of predicting unknown source code units within a software system. Situations such as the addition of new features or source code units fits this experiment.

The class-level training and prediction accuracy of unknown data within a system is shown in Figure 4.9. We can see that a number of the test subjects have large variations in their accuracy, and in four cases actually hit 0%. These 0% situation indicate that nothing was correctly predicted, we how this situation might be occurring

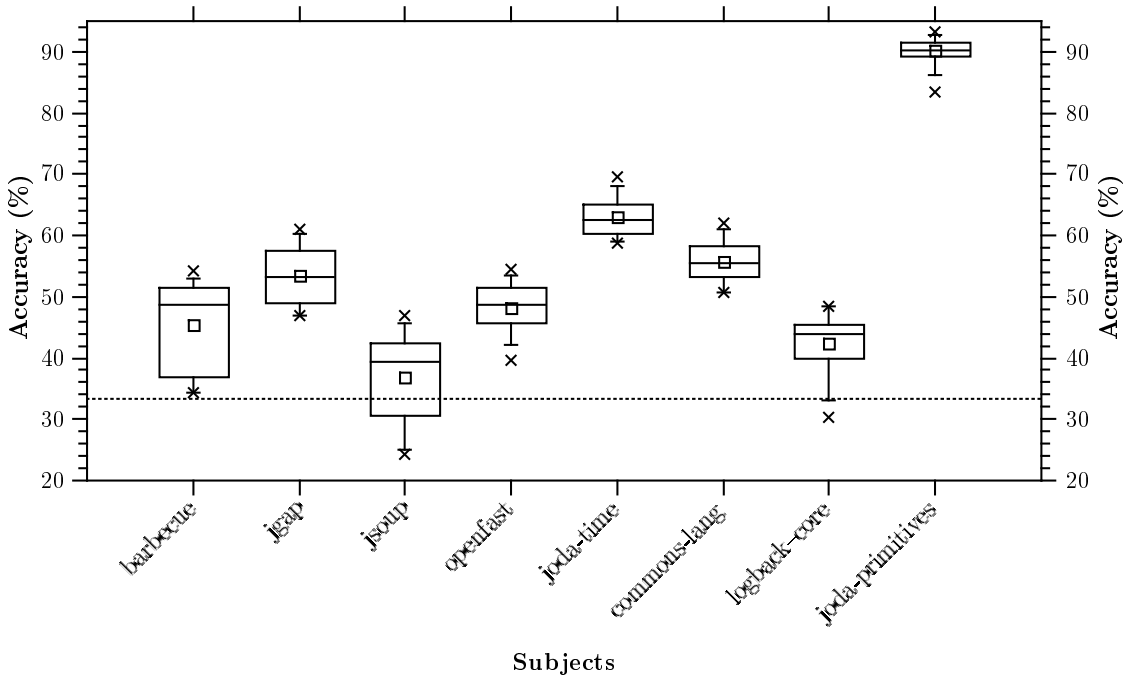


Figure 4.10: Method-level training and prediction accuracy on unknown data within a system.

in Section 4.3.4. We also can note that half of the test subjects for class-level prediction of unknown data within a system had a performance worse than random. The average prediction accuracy for this experiment is 31.7%±10.2%, which is lower than random with a large standard deviation.

The method-level training and prediction accuracy of unknown data within a system is shown in Figure 4.10. The mean average prediction accuracies of the test subject vary, which suggests that the prediction quality might depending on the project itself, or that more data is required for this experiment. While *joda-primatives*'s mean prediction accuracy for class-level was 0%, the mean accuracy for method-level was approximately 90%. It seems to be that *joda-primatives* represents an outlier in our test subjects. All mean prediction accuracies exceed random for method-level

predictions, while class-level predictions did not fare as well. This suggests that class-level predictions are harder to predict because:

- Classes have much more factors involved in them (i.e., a set of methods) thus harder to predict.
- Our approach does not account for overloaded, anonymous, and abstract methods thus the classes are partially incomplete in data.
- For our experiment we had considerably less class-level data available than that of method-level data.

The average prediction accuracy for this experiment is 53.2%±5.1%, which is higher than random with half the standard deviation of the class-level predictions.

4.3.3.2 Prediction Across Systems

This experiment explores the capability of predicting unknown source code units across a software system. Whether predictions will fare as well as predictions within a system (see Section 4.3.3.1 is explored in this section.

The class-level training and prediction accuracy of unknown data across a system is shown in Figure 4.11. Of the eight test subjects three of them are below random with respect to mean prediction accuracy. With respect to Figure 4.9, the class-level prediction accuracy within a system has much more variation in accuracy than across systems. The less variation in prediction accuracy across systems is most likely due to the fact of having more data items present as we now consider seven of the test subjects for training data. The average prediction accuracy for this experiment is 34.4%±4.7%, which is slightly higher than random and is an improvement over the class-level within systems (only because the within systems had a 0% for *joda-primatives*, otherwise within a system average accuracy would have been 36.2%).

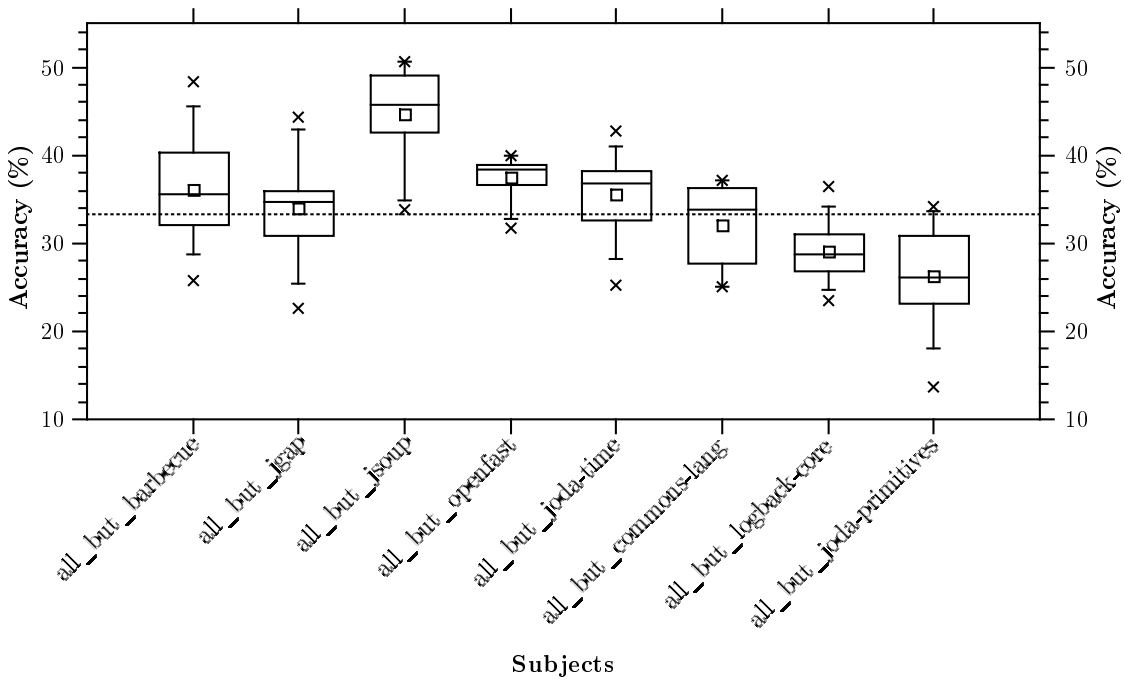


Figure 4.11: Class-level training and prediction accuracy on unknown data across systems.

The method-level training and prediction accuracy of unknown data across a system is shown in Figure 4.12. Of the test subjects all but *joda-primatives* have a mean accuracy that exceeds random. With respect to Figure 4.10, the method-level prediction accuracy within a system has much more variation in accuracy than across systems. The less variation in prediction accuracy across systems is most likely due to the fact of having more data items present as we now consider seven of the test subjects for training data. The average prediction accuracy for this experiment is 37.6%±2.6%, which is slightly higher than random. With respect to the prediction accuracy within a system for method-level source code units this experiment has shown a drop of 15.6% in prediction accuracy. This drop in accuracy suggests that prediction across systems is a more challenging prediction to make regarding mutation score prediction.

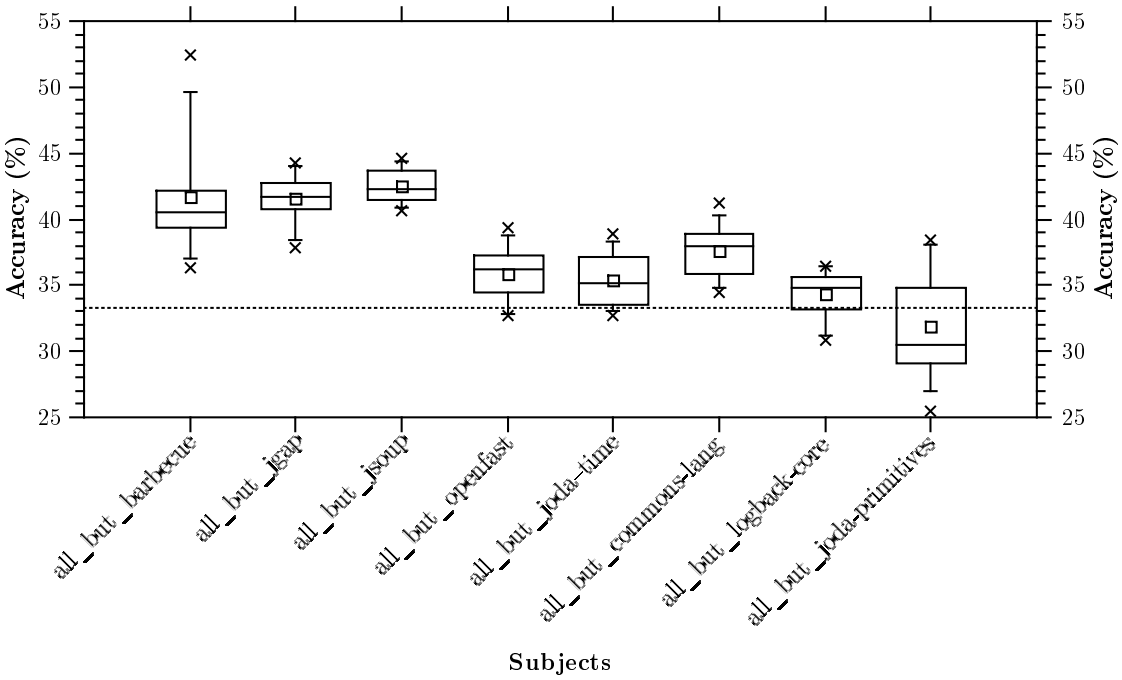


Figure 4.12: Method-level training and prediction accuracy on unknown data across systems.

4.3.4 Optimization and Generalization

> ***Research Question #1:*** *Can we optimize our approach to achieve better performance by using a different measure of classifier performance?*
>
> ***Research Question #2:*** *Can we identify a general set of SVM parameters that maximize mutation score prediction performance on unknown data?*

We kept track of the frequency of parameter pairs selected over all the prediction experiments conducted in Section 4.3.3. As a result of *LIBSVM*'s *script* for parameter selection, we saw 57 different pairings of the RBF kernel parameters *cost* and *gamma* (described in Section 2.2.2). This indicates that the classifiers are being tuned specifically to maximize the cross-validation accuracy. Due to undersampling, different

parameters are being used to ensure a maximization of cross-validation accuracy. To encourage generalization of unknown projects and data, ideally we want to find a parameter pairing that maximizes generalizability, effectively the classification performance on unknown data. As our approach initially requires mutation testing results to perform training it might be appropriate to select the best parameters for the given data. In the previous section for predictions on unknown data (see Section 4.3.3), we explored predictions within and across systems. For this experiment we continue with the same setup by considering across and within systems. Regarding this experiment it becomes much more clear on why we need a generalizable set of parameters that hopefully perform well on most test subjects. In terms of usability, if we can find a general set of parameters for predicting mutation scores, this will lessen the need to specifically tune every classifier prior to prediction.

We noticed that in some situations accuracy is not the best measure for a classifier's effectiveness. For example, given imbalanced data for the testing/unknown data set, the accuracy could misrepresent the performance of the classifier. The raw outputs of our classifier using two different sets of parameters on the *joda-time* subject are shown in Figures 4.13 and 4.14. In both of the raw outputs we can see a confusion matrix along with performance measures. We can see in the raw output in Figure 4.13 that all predictions fall in category 2. The *joda-time* data set is imbalanced with the majority of data (i.e., 76.2% of the data) belonging to category 2. Even with the biased predictions made towards the majority category, the accuracy of the prediction is 76.2%. In contrast to the the raw output presented in Figure 4.14 we can see that the accuracy is slightly lower at 71.7%. Now even though the accuracy in the second example is slightly lower we can consider it a superior classifier to the former as it actually treats the categories in a more unbiased fashion (i.e., one category is not receiving the majority of predictions like the previous example). To alleviate this

Figure 4.13: Raw output of training on *joda-time* then predicting on its unknowns using the parameters *cost*=0.03125 and *gamma*=0.0078125.

	P0	P1	P2
A0	0	0	0
A1	0	0	296
A2	0	0	946

	TPR	FPR	Accuracy	Recall	Precision	Specificity	F-score	Bal. Acc.	Youden-index
A0	0.000000	0.000000	0.761675	0.000000	0.000000	1.000000	0.000000	0.500000	0.000000
A1	0.000000	0.000000	0.761675	0.000000	0.000000	1.000000	0.000000	0.500000	0.000000
A2	1.000000	1.000000	0.761675	1.000000	0.761675	0.000000	0.864717	0.500000	0.000000

Figure 4.14: Raw output of training on *joda-time* then predicting on its unknowns using the parameters *cost*=8 and *gamma*=0.125.

	P0	P1	P2
A0	0	0	0
A1	23	180	93
A2	55	180	711

	TPR	FPR	Accuracy	Recall	Precision	Specificity	F-score	Bal. Acc.	Youden-index
A0	0.000000	0.080495	0.717391	0.000000	0.000000	0.919505	0.000000	0.459752	-0.080495
A1	0.608108	0.202020	0.717391	0.608108	0.500000	0.797980	0.548780	0.703044	0.406088
A2	0.751586	0.340659	0.717391	0.751586	0.884328	0.659341	0.812571	0.705463	0.410926

Performance Measure	Bad Classifier (Figure 4.13)	Good Classifier (Figure 4.14)
Mean Accuracy	76.2%	71.7%
Mean F-score	28.8%	45.4%
Mean Balance Accuracy	50.0%	62.3%
Mean Youden-index	00.0%	24.6%

Table 4.7: Comparison of performance measures for a *bad* classifier vs. a *good* classifier.

problem, we consider other measurements that can be used to assess the predictive capabilities of classifiers, specifically the following measures:

- **F-score** represents the harmonic mean of the recall and precision for a category [SJS06]. A score closer to 1 (i.e., 100%) represents better performance.

$$F\text{-}score = 2 * \frac{recall * precision}{recall + precision} \tag{4.1}$$

- **Balanced Accuracy** represents the average accuracy obtained on the category [BOSB10, SJS06]. A score closer to 1 (i.e., 100%) represents better performance.

$$balanced\ accuracy = \frac{recall + specificity}{2} \tag{4.2}$$

- **Youden's Index** represents the classifier's ability to avoid failure [SJS06]. It can also be calculated using the balanced accuracy. A score closer to 1 (i.e., 100%) represents better performance.

$$youden's\ index = recall - (1 - specificity) \tag{4.3}$$

$$youden's\ index = 2 * balanced\ accuracy - 1 \tag{4.4}$$

Using their accuracy, the *bad* and *good* classifiers were unable to distinguish the better classifier (i.e., fair predictions over all categories), while the three aforementioned performance measures are capable of doing so. We take the average value of the performance measures (i.e., the sum divided by the three categories for each measure) and compare classifiers in Table 4.7. The comparison shows that the new performance measures better reflect the performance of the classifier than traditional accuracy in all three cases.

To further generalize our predictions, we conducted our own grid search with comparisons to prediction accuracy instead of cross-validation accuracy. As briefly mentioned in Section 4.1.1 a grid search performs a search over a range of parameters, which in our case is *cost* and *gamma*. We use a coarse search over the parameter ranges between 0.00001 and 10000 by adjusting the order of magnitude by a factor of ten. The following outlines our strategy to find the pairing of parameters that maximizes the performance of our SVM on predicting unknown data:

1. Grid search using coarse parameter ranges between 0.00001 and 10000 by adjusting the order of magnitude by a factor of ten.

2. We maximize the F-score (we could have used Balance Accuracy or Youden-index) on unknown data (i.e., what remains after undersampling or the excluded test subject) instead of on cross-validation of the training data.

3. We conduct the previous two steps (i.e., grid search of an data set) on each of the individual test subjects and also for the *all_but_*<*subject*> subjects. For each test subject we perform ten executions for each parameter pairing to account for undersampling.

4. Use a simple rank summation (i.e., ascending rank n has a value of n) to tally the parameter pairings that consistently performed the best on the data sets.

5. We pick the parameter pairing that perform best on both the individual subjects and the *all_but_<subject>* subjects.

Using our search strategy as described we attained the following SVM parameters for class-level [$cost$=100, $gamma$=0.01] and method-level [$cost$=100, $gamma$=1]. These parameters were found to offer the greatest generalizable over the different test subjects. Furthermore, by maximizing F-score instead of accuracy these parameters will avoid issues presented by using accuracy alone. Section 4.3.4.1 presents results for prediction within a system, and Section 4.3.4.2 presents results for prediction across systems. Both which utilize the found set of parameters that offer the greatest performance with respect to maximizing F-score on predicting unknown data. A comparison between pre/post generalized parameter prediction performance is explored in Section 4.3.4.3.

4.3.4.1 Prediction Within a System

This experiment is the same as the previous one (see Section 4.3.3.1) in that we are concerned with assessing the prediction capability within a system. The only difference is that these results are based on a generalized set of parameters found through a grid search (found in Section 4.3.4).

The class-level training and prediction accuracy of unknown data within a system using generalized parameters is shown in Figure 4.15. We can see that a number of the test subjects still have large variations in their accuracy when compared to the same experiment without the generalized parameters (see Figure 4.9). There are no more cases of 0% mean accuracy anymore, which is a good sign that the generalized parameters using F-score actually alleviated this situation. Three of the eight subjects have a mean accuracy lower than random. The average prediction accuracy for this experiment is 36.7%±10.1%, which is slightly higher than random

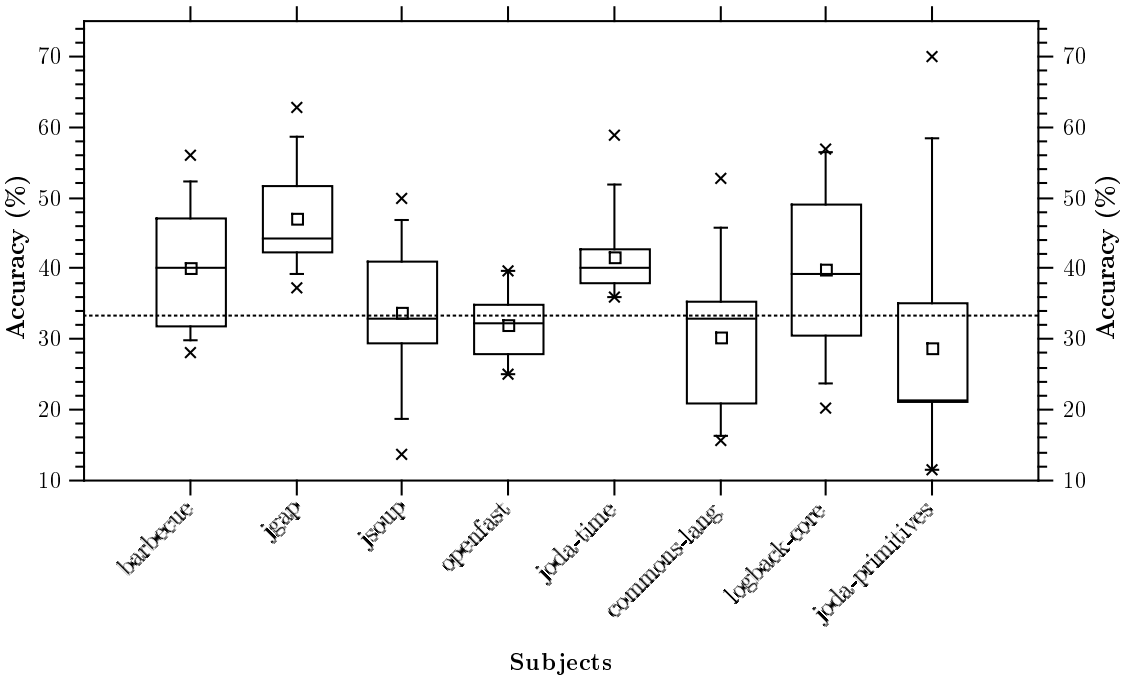


Figure 4.15: Class-level training and prediction accuracy on unknown data within a system using generalized parameters [*cost*=100, *gamma*=0.01].

and is an improvement of +5.0% over the non-generalized parameter experiment (see Table 4.8).

The method-level training and prediction accuracy of unknown data within a system using generalized parameters is shown in Figure 4.16. We can see that the results are similar when compared to the same experiment without the generalized parameters (see Figure 4.10). The average prediction accuracy for this experiment is 56.8%±6.2%, which is higher than random and is an improvement of +3.6% over the non-generalized parameter experiment (see Table 4.10).

4.3.4.2 Prediction Across Systems

This experiment is the same as the previous one (see Section 4.3.3.2) in that we are concerned with assessing the prediction capability across systems. The only difference

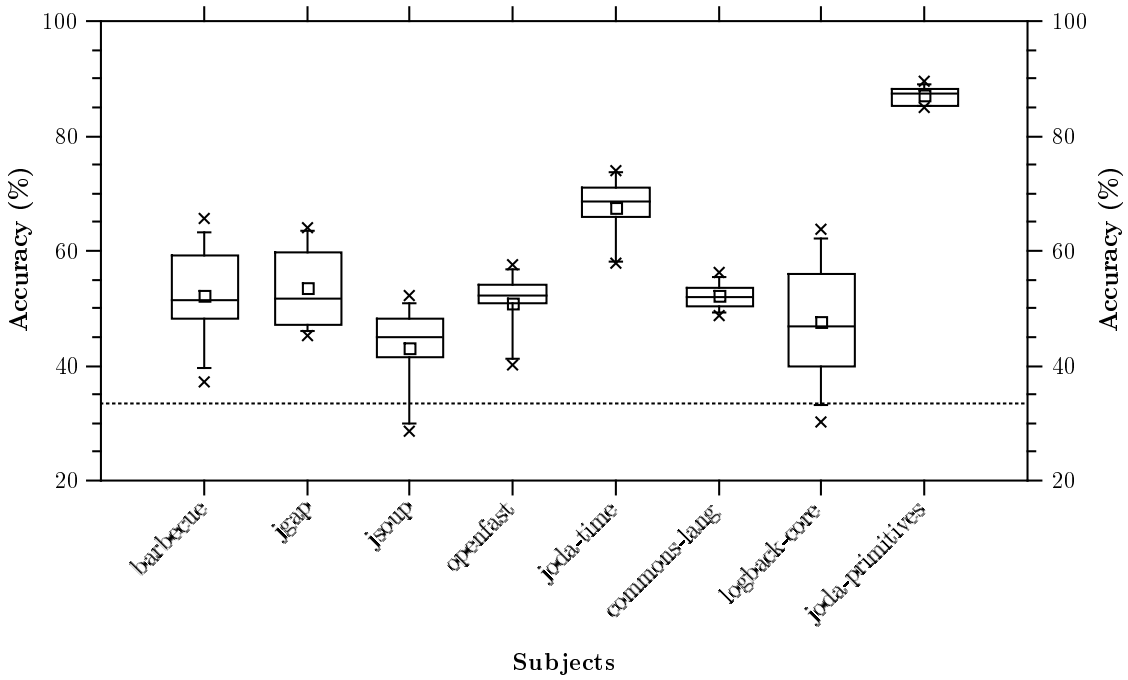


Figure 4.16: Method-level training and prediction accuracy on unknown data within a system using generalized parameters [$cost$=100, $gamma$=1].

is that these results are based on a generalized set of parameters found through a grid search (found in Section 4.3.4).

The class-level training and prediction accuracy of unknown data across systems using generalized parameters is shown in Figure 4.17. When compared to the same experiment without the generalized parameters (see Figure 4.11) we can see slight improvements in terms of mean accuracy, in particular there are only two test subjects with less than random mean accuracy. The average prediction accuracy for this experiment is 39.0%±3.9%, which is slightly higher than random and is an improvement of +4.6% over the non-generalized parameter experiment (see Table 4.9).

The method-level training and prediction accuracy of unknown data across systems using generalized parameters is shown in Figure 4.18. We can see that the results are similar when compared to the same experiment without the generalized parameters

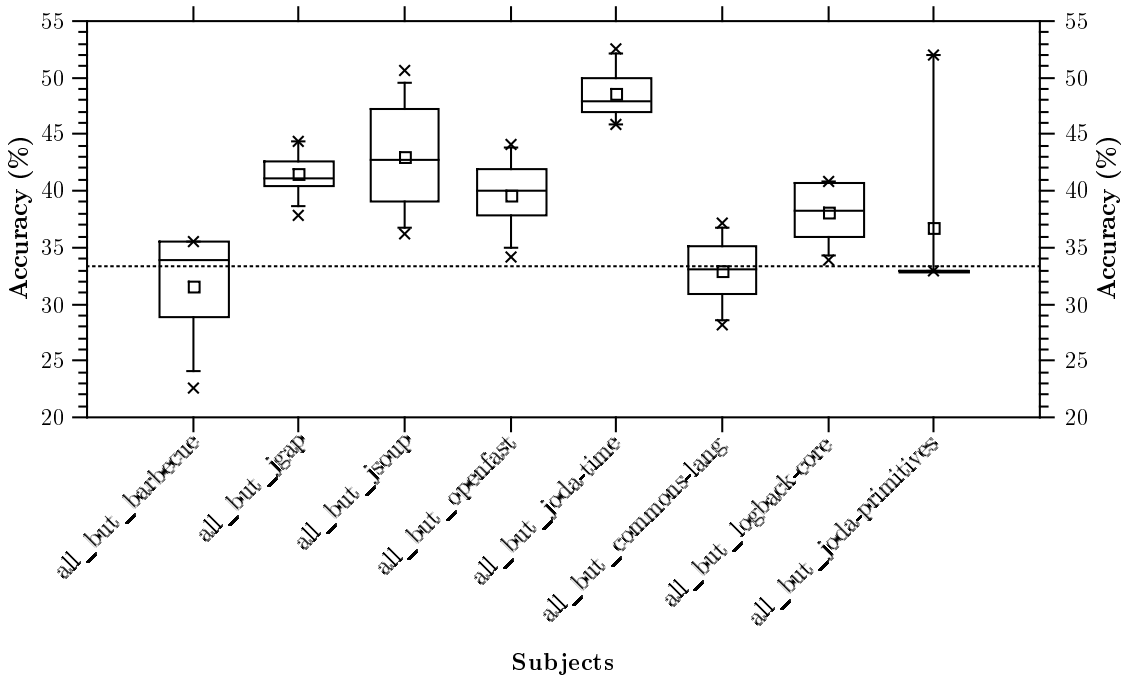


Figure 4.17: Class-level training and prediction accuracy on unknown data across systems using generalized parameters [*cost*=100, *gamma*=0.01].

(see Figure 4.12). The average prediction accuracy for this experiment is 42.8%±1.8%, which is higher than random and is an improvement of +5.2% over the non-generalized parameter experiment (see Table 4.11).

4.3.4.3 The Effects of Generalized Parameters on Prediction Performance

Using the generalizable parameters we can see that in both class- and method-level results, the resulting accuracy usually increases slightly. In some situations we can even see a decreased in the standard deviation for accuracy. In particular, we see that in class-level predictions, the possibilities of 0% accuracy (which occurred in four of the test subjects without the generalizable parameters) no longer occurs. This happened as a result of selecting parameters that maximized F-score instead of cross-validation accuracy. This change treats the predictions of categories more fairly (i.e., avoiding

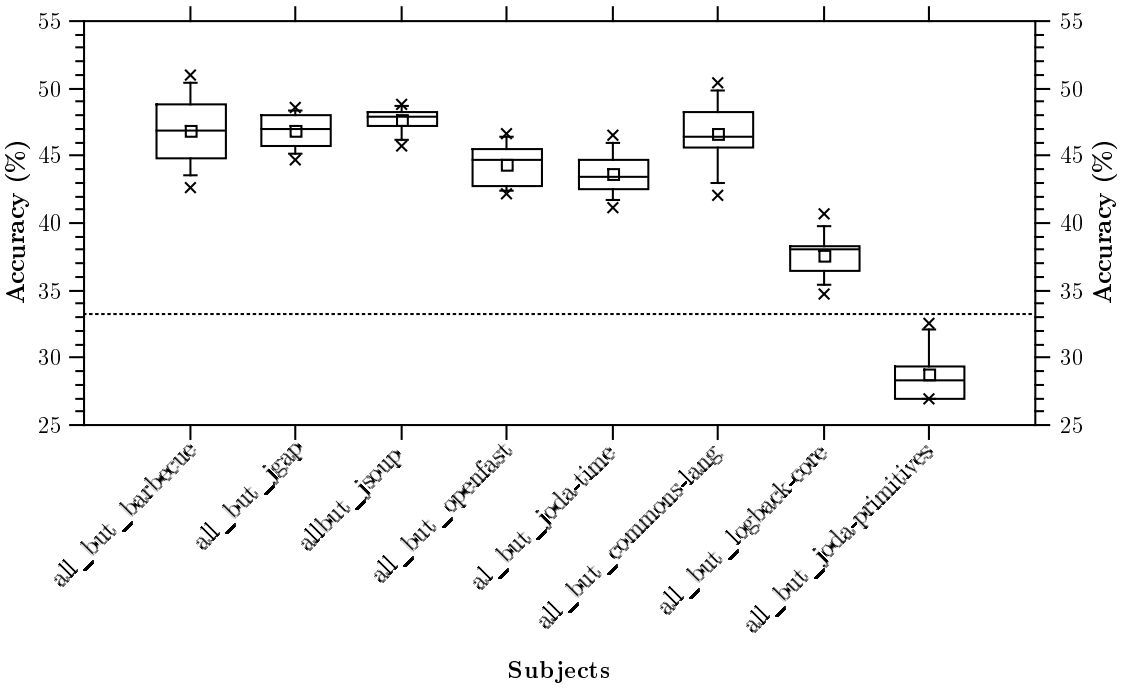


Figure 4.18: Method-level training and prediction accuracy on unknown data across systems using generalized parameters [$cost$=100, $gamma$=1].

predicting all of one category, which could be the wrong category). To further see the benefits of using generalized *LIBSVM* parameters we compared the individual accuracies and standard deviation of each test subject.

As presented in Tables 4.8 – 4.11, we can see the gains and losses in mean and standard deviation of prediction accuracy resulting from the application of generalized parameters. In terms of comparative changes, an improvement for mean accuracy would be a gain (i.e., better prediction accuracy) while for standard deviation an improvement would be a loss (i.e., smaller variation in prediction accuracy). Of the 16 class-level test subjects presented in Table 4.8 and 4.9, 12 out of 16 test subjects saw an improvement in mean accuracy and 9 out of 16 test subjects saw an improvement in standard deviation. Of the method-level test subjects presented in Table 4.10 and 4.11, 13 out of 16 test subjects saw an improvement in mean accuracy and 10 out 16 test

Table 4.8: Comparison of class-level prediction accuracy within systems (mean ± standard deviation) before/after generalized parameters are used.

Test Subject	Before Parameter Generalization (%) (Figure 4.9)	After Parameter Generalization (%) (Figure 4.15)	Gain(↑)/Lost(↓) from Parameter Generalization (%)
logback-core	54.3±3.9	39.9±12.5	↓14.4±↑8.6
barbecue	34.4±15.5	40.0±9.2	↑5.6±↓6.3
jgap	38.1±20.5	47.0±7.5	↑8.9±↓13.0
commons-lang	27.0±17.9	30.3±11.3	↑3.3±↓6.6
joda-time	42.7±6.2	41.6±6.6	↓1.1±↑0.4
openfast	28.1±5.1	32.1±5.3	↑4.0±↑0.2
jsoup	28.6±12.6	33.9±10.4	↑5.3±↓2.2
joda-primitives	0.0±0.0	28.9±17.7	↑28.9±↑17.7
average	**31.7±10.2**	**36.7±10.1**	**↑5.0±↓0.1**

Table 4.9: Comparison of class-level prediction accuracy across systems (mean ± standard deviation) before/after generalized parameters are used.

Test Subject	Before Parameter Generalization (%) (Figure 4.11)	After Parameter Generalization (%) (Figure 4.17)	Gain(↑)/Lost(↓) from Parameter Generalization (%)
all_but_logback-core	29.0±3.7	38.1±2.7	↑9.1±↓1.0
all_but_barbecue	36.1±6.4	31.6±4.8	↓4.5±↓1.6
all_but_jgap	34.0±6.1	41.5±2.0	↑7.5±↓4.1
all_but_commons-lang	32.1±2.1	32.9±3.0	↑0.8±↑0.9
all_but_joda-time	35.6±4.9	48.6±2.3	↑13.0±↓2.6
all_but_openfast	37.4±2.6	39.7±3.2	↑2.3±↑0.6
all_but_jsoup	44.7±5.8	43.0±5.0	↓1.7±↓0.8
all_but_joda-primitives	26.3±6.1	36.7±8.1	↑10.4±↑2.0
average	**34.4±4.7**	**39.0±3.9**	**↑4.6±↓0.8**

Table 4.10: Comparison of method-level prediction accuracy within systems (mean ± standard deviation) before/after generalized parameters are used.

Test Subject	Before Parameter Generalization (%) (Figure 4.10)	After Parameter Generalization (%) (Figure 4.16)	Gain(↑)/Lost(↓) from Parameter Generalization (%)
logback-core	42.4±5.7	47.6±10.9	↑5.2±↑5.2
barbecue	45.4±8.0	52.3±8.7	↑6.9±↑0.7
jgap	43.4±5.1	53.4±7.2	↑10.0±↑2.1
commons-lang	55.8±3.7	52.2±2.3	↓3.6±↓1.4
joda-time	63.0±3.4	67.5±5.6	↑4.5±↑2.2
openfast	48.3±4.3	51.0±5.6	↑2.7±↑1.3
jsoup	36.8±8.0	43.2±7.6	↑6.4±↓0.4
joda-primitives	90.1±2.7	87.1±1.5	↓3.0±↓1.2
average	**53.2±5.1**	**56.8±6.2**	**↑3.6±↑1.1**

Table 4.11: Comparison of method-level prediction accuracy across systems (mean ± standard deviation) before/after generalized parameters are used.

Test Subject	Before Parameter Generalization (%) (Figure 4.12)	After Parameter Generalization (%) (Figure 4.18)	Gain(↑)/Lost(↓) from Parameter Generalization (%)
all_but_logback-core	34.3±1.9	37.6±1.7	↑3.3±↓0.2
all_but_barbecue	41.7±4.6	46.9±2.6	↑5.2±↓2.0
all_but_jgap	41.6±2.0	46.9±1.3	↑5.3±↓0.7
all_but_commons-lang	37.6±2.1	46.6±2.5	↑9.0±↑0.4
all_but_joda-time	35.4±2.1	43.7±1.6	↑8.3±↓0.5
all_but_openfast	35.9±2.2	44.3±1.6	↑8.4±↓0.6
all_but_jsoup	42.5±1.3	47.7±1.0	↑5.2±↓0.3
all_but_joda-primitives	31.9±4.2	28.8±2.0	↓3.1±↓2.2
average	**37.6±2.6**	**42.8±1.8**	**↑5.2±↑0.8**

subjects saw an improvement in standard deviation. These results show that the generalized parameters overall had a positive effect on the test subjects's performance.

Overall, the following summarizes the results of using generalized parameters for our approach on prediction of unknown data within and across systems:

- Class-level average prediction accuracy on unknown data within systems saw an improvement of +5.0% with an improvement of -0.1% in standard deviation. Resulting in a average prediction accuracy of 36.7%±10.1%.

- Class-level average prediction accuracy on unknown data across systems saw an improvement of +4.6% with an improvement of -0.8% in standard deviation. Resulting in a average prediction accuracy of 39.0%±3.9%.

- Method-level average prediction accuracy on unknown data within systems saw an improvement of +3.6% with a decline in standard deviation of +1.1. Resulting in a average prediction accuracy of 56.8%±6.2%.

- Method-level average prediction accuracy on unknown data within systems saw an improvement of +5.2% with a decline in standard deviation of +0.8. Resulting in a average prediction accuracy of 42.8%±1.8%.

With respect to predictions in general, method-level prediction have a higher mean accuracy in all situations (i.e., within and across systems). Furthermore, predictions within systems tend to have higher standard deviation than predictions across systems. This difference in standard deviation most likely is attributed to the abundance of data items present for training and prediction. With respect to prediction accuracy the class-level predictions actually performed better across systems only by a difference of 2.3%, while method-level predictions performed better within systems by a difference of 14.0%.

The improvements in both class- and method-level are both a side benefit of using generalized *LIBSVM* parameters, as the main purpose was to nullify the need for parameter selection (i.e., no need to grid search on known data) to make predictions on unknown data. After optimizations and generalization, our approach for mutation score prediction using source code and test suite metrics can out perform random in nearly all test subjects observed.

4.3.5 Impact of Training Data Availability on Prediction Accuracy

> ***Research Question #1:*** *How is the prediction accuracy impacted by the availability of training data?*
>
> ***Research Question #2:*** *Is it possible to only train on a fraction of the source code units and achieve approximately the same prediction performance on the remaining source code units?*

As we saw in the previous section using our approach we achieve 56.8% prediction accuracy of method-level source code units within systems. This result exceeds random by 23.5% and therefore shows that our approach is capable of predicting mutation score categories using source code and test suite metrics at least within systems. As mentioned in the thesis statement, *"The predictions can be used to reduce the resource cost of mutation testing in traditional iterative development."*. Iterative development is a software development life cycle that allows developers to work on small changes which eventually adds to a large change with respect to a software system. Traditional iterative development may involve expanding/reducing/refactoring the SUT, and/or attempting to improve the test suite. By including mutation testing between iterations to determine if any improvements have occurred can be costly if done in a naive manner

(i.e., re-conduct the whole mutation test process using the new version of the SUT). Even with an intelligent approach of selective mutation (i.e., only mutation testing source code units that were added/removed/modified since the previous iteration), the cost of mutation testing can still be substantial.

There is no consensus on good ratios for training data and testing data. Using the common 10-fold cross-validation [Koh95] as a guideline, a 9:1 ratio for training data to testing data is a good rule-of-thumb. Using 90% of a large data set for training might be considered wasteful when considering a classifier's *learning curve*. Provost et al. mention *"Learning curves typically have a steeply sloping portion early in the curve, a more gently sloping middle portion, and a plateau late in the curve"* [PJO99]. Using a representative sample of the available training data should retain nearly the same predictive performance with respect to supervised learning. Through an empirical evaluation, Provost et al. demonstrated that the minimum amount of training data required to maximize the prediction performance of a classifier (i.e., reaching the plateau of the learning curve) varies based on the data set. Using a progress random sampling algorithm, Provost et.al. were able to determine the minimum amount of training data for their data sets, they concluded that the minimum amount of data required is different for each data set. From their experimentation with three data sets they found that the minimum percent of training data required for maximum prediction performance was 2% (of 100000 items), 12% (of 100000 items) and 25% (of 32000 items). Finally, Provost et al. were able to demonstrate the benefit of random sampling to reduce the training set, namely a reduction in computational cost using a smaller training data set. Wang et al. also demonstrated random sampling reduce the amount of training data necessary to achieve maximum prediction performance [WNC05]. Wang et al. evaluated a range of random sampling (ranging

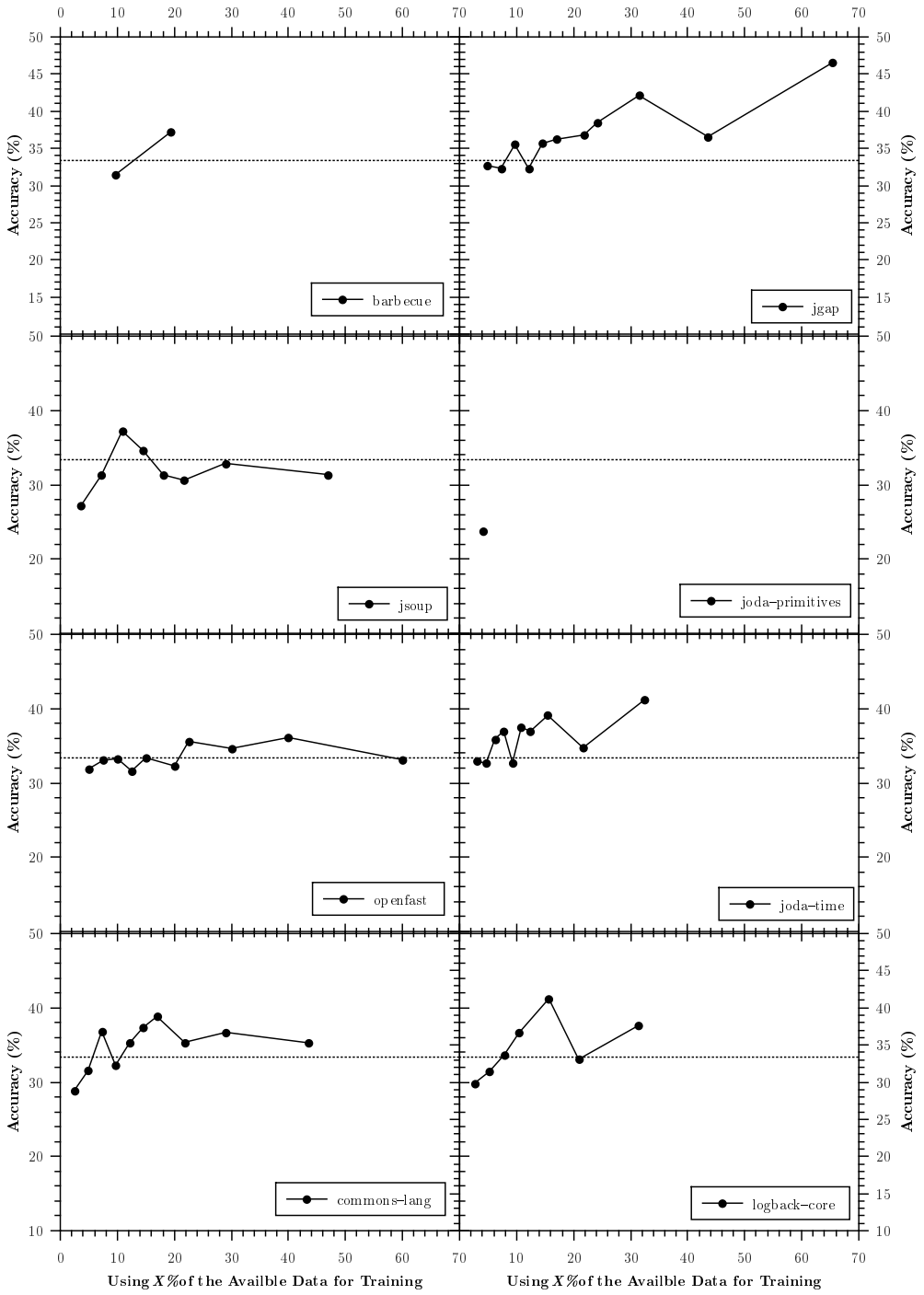


Figure 4.19: Class-level prediction accuracies of each test subject using training and prediction with various amounts of training data.

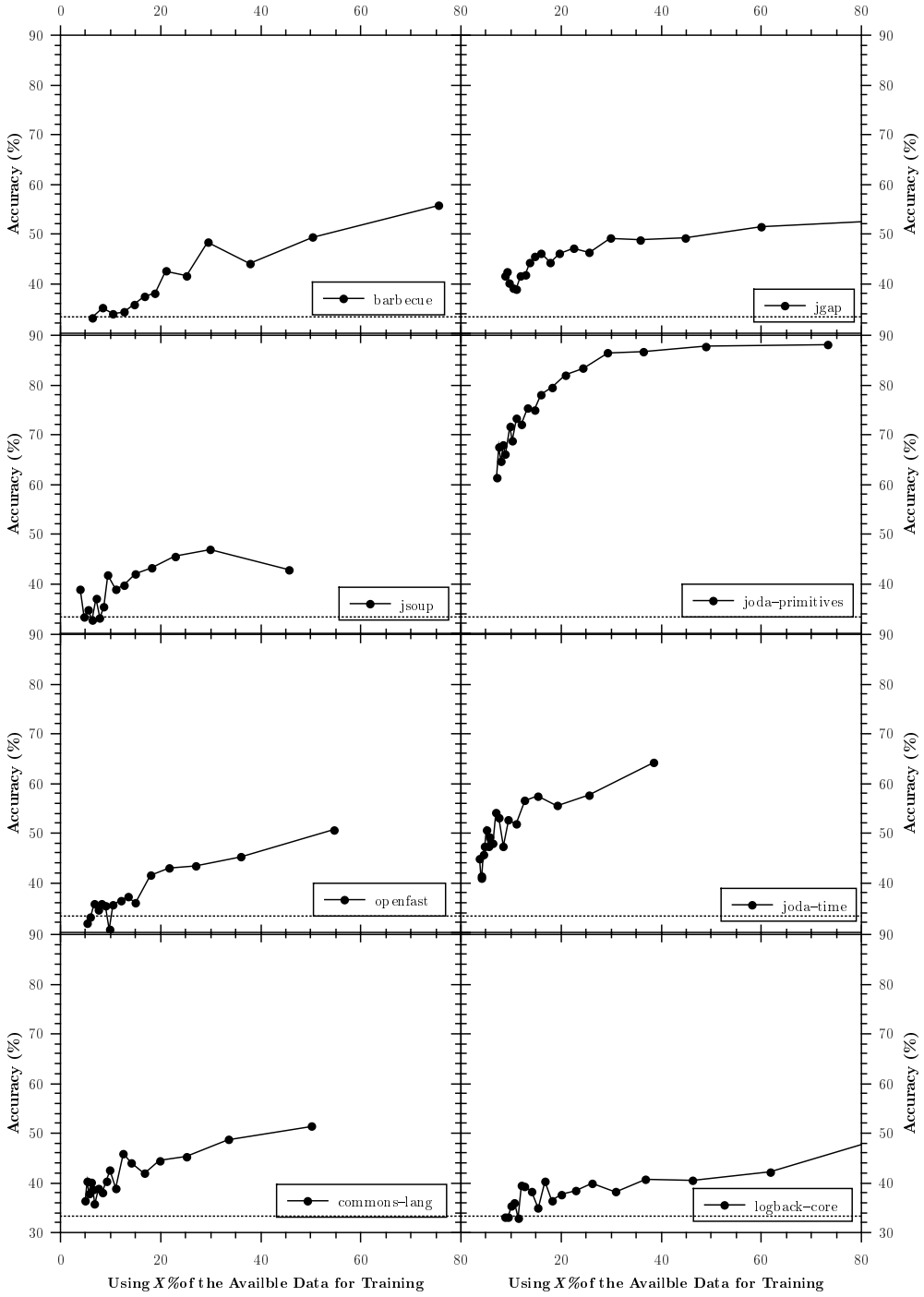


Figure 4.20: Method-level prediction accuracies of each test subject using training and prediction with various amounts of training data.

from 10%-100%) of four data sets and the greatest loss in prediction performance was approximately 10.5%.

Random sampling appears to perform well for reducing the training set size while retaining prediction performance [PJO99, WNC05]. There exist more complex approaches to this problem, one uses centroids of weighted clusters which essentially groups similar items in the training set and treating them as one item [NBP08]. With our approach with respect to data availability, we are interested in minimizing the data required to make accuracy predictions. Similar to the previous research on random sampling to reduce the training data set, we decided to explore how it affects our prediction accuracy. Ideally we can reduce the resource cost of mutation testing in traditional iterative development with intermixed iterations of mutation testing and predictions by performing mutation testing on a portion of the SUT and predicting the remaining portion.

We conducted a number of training and prediction executions using different amounts of source code units for training. We took the amount of undersampled training data points and divided this amount by intervals of 0.5 from 1.0 to 10.0. We conducted ten executions using the generalized parameters from Section 4.3.4 for each new training amount and recorded the mean accuracy. In situations where the new training amount was identical to another's interval their resulting accuracy were averaged. As we can see in Figure 4.19, the class-level source code units did not show much information as there was a limited number of unique points for the test subjects. This is due to the limited data set available for the test subjects, recalling *barbecue* only had two data points per category, while *joda-primitives* only had one. Unfortunately there does not seem to be enough data to warrant any observation from the class-level. With Figure 4.20 we can see an apparent trend for method-level source code units, and there appears to be a $log(n)$ relationship with prediction accuracy

and the amount of data used for training (i.e., the *learning curve*). *joda-primitives* shows exactly the trend were expecting: the prediction accuracy tapers off reaching its maximum value between 30%–35%. Looking at other test subjects we can see similar behaviour, though not as pronounced.

We know that there exists a minimum number of training data points required to reach the prediction accuracy plateau, however with our data sets we might not have enough data to see this effect. Considering that the previous research works see results with a little as 2% of the training data used, we cannot possibly achieve results like that considering we have less than 1000 items for our individual test subjects comparative to 100000 items. A quick observation of our results suggests that we could probably use a fraction of the available data from a SUT to achieve near optimal prediction accuracy. In our case we would suggest using one third or more of the available data for training purposes to maximize prediction accuracy. By considering a fraction of the mutants for training purposes it is not necessary to evaluate the remaining fraction and our approach could be used to predict the mutation score of the remaining source code units. To account for the potential mis-classifications (considering we have approximately 50% prediction accuracy) it would be best to cycle the training data such that the we select a new random sample that is mostly unique in each iteration. With enough iterations all mutants would have been actually evaluated once, while only really evaluating a fraction of the mutants from the SUT. As a side effect, we can assume that at some point all mutants have actually been evaluated and we could keep this information in our database and reuse it for training purposes.

4.4 Threats to Validity

We consider the four categories for threats to validity with respect to experimentation in Section 4.4.1 to 4.4.4.

4.4.1 Conclusion Validity

Threats to conclusion validity involve issues with the process and statistical means to draw any conclusions regarding experiments [WRH+00, WKP10]. We utilized various summary statistical measures to determine the conclusions of our results. In particular, we used mean, standard deviation, quartiles and frequencies to understand our experiments with respect to their results. Furthermore, with our results we conducted a minimum of ten executions per experiment to mitigate the randomness of our results. With respect to drawing conclusions, we compared the average accuracy to what a random prediction would achieve. Thus by comparing the mean accuracies we were able to compare our approach to random. In retrospect, we should have performed more executions per experiment to further reduce the noise. Furthermore we could have performed a statistical test to understand the statistical significants of our comparison.

4.4.2 Internal Validity

Internal threats to validity are concerned with factors that could influence the independent variable in our experiments [WRH+00, WKP10]. Our independent variables are the features themselves from the eight open source software systems that we selected. Obviously there could be issues that can arise based on the measures that our tools returned for each software system, though these tools are well established and provide simplistic measures (i.e., issues are unlikely to arise due to *incorrect* results). With

respect to the mutants themselves that are generated by the *Javalanche* mutation testing tool, the version used was experimental and could be more susceptible to *incorrect* results. Furthermore *Javalanche* uses a subset of method-level mutation operators, which could have a major impact on the class-level source code units. With respect to *true* internal validity the independent variables are not influencing each other in ways that we were not aware of that could be detrimental to our experiment.

4.4.3 Construct Validity

Whether the independent and dependant variables we are using actually align with the problem with which we are experimenting is an issue with construct validity [WRH+00, WKP10]. In our experiment we are using a set of features extracted from open source software systems (i.e., the independent variables) to determine the accuracy of predicting mutation score (i.e, the dependant variable). Machine learning performance measures (i.e., accuracy, F-score, etc. . .) are valid dependant variables as they measure the effectiveness of the classification technique. The independent variables for machine learning are harder to determine by nature, there is often no clear set of features for making predictions. For our experiment, we observed the two main components involved in mutation testing, the source code and test suite. These two components can be represented in quantifiable metrics (i.e., source code and test suite metrics) which are commonly known and used in Software Engineering research.

4.4.4 External Validity

With experiments, one of the major concerns involves the ability for the results to generalize outside of the study. That is external validity [WRH+00,WKP10]. With our experiment we specifically avoided toy-problems and opted to use open source software

systems, which are real software systems. These software systems are not industrial nor are they extremely large-scale (i.e., 100000+ SLOC), thus we are unsure if the results would generalize to such software systems. The test subjects we chose had some variation in domain (i.e., library, framework, etc...) though our set obviously does not act as a representative of different domains. In addition, most of the test subjects we used had relatively *good* test suites (i.e., of the covered mutants the mutation scores were above 73% except for one test subject). Due to this, we are unsure how our prediction would perform on software systems with *poor* test suites. Furthermore we observed only the Java language, whether these results generalize to other languages has not been verified. As stated by Kitchenham and Mendes *"It is invalid to select one or two data sets to 'prove' the validity of a new technique because we cannot be sure that, of the many published data sets, those chosen are the only ones that favour the new technique"* [KM09]. We used only eight open source projects as our data sets for our prediction technique, however even though this is more then one or two it is still quite limited. Mutants can be influenced by external factors such test suite size and mutation operators as it was found that class-level mutants are harder to detect than traditional method-level mutants [NK11]. As we used only traditional mutation operators this could have an impact on the generalizability, along with the varying sizes of the test suites of our test subjects.

Recall that our approach for predicting mutation scores based on source code and test suite metrics utilizes a number of tools:

- *Javalanche* to collect mutation scores.
- *Eclipse Metrics Plugin* to collect source code and test suite metrics.
- *EMMA* to collect additional test suite coverage metrics.
- *LIBSVM* to perform the training and prediction of the source code units.

We selected tools based on the metrics they could provide as well as the the output format, yet there might be other tools that could have performed better. In particular, the mutation testing tool we selected is not the newest, and omits a whole class of mutations (i.e., class-level object-oriented mutants), which could be misrepresenting the mutation scores. The tools used to collect the features of the source code units might not be comprehensive in terms of features that describe the source code units.

4.5 Summary

In this chapter we covered the following topics that demonstrate out approach on several test subjects:

- In Section 4.1 we covered our experimental setup with respect to environment, test subjects, tool configuration and data preprocessing.
- In Section 4.2 we discussed our experimental method for the five experiments that were conducted in this chapter.
- In Section 4.3 we covered a number of experiments and discussed their results. Specifically we experimented with mutation score distribution (Section 4.3.1), cross-validation (Section 4.3.2), prediction (Section 4.3.3), optimization and generalization (Section 4.3.4), and the impact of data availability (Section 4.3.5).
- In Section 4.4 we discussed conclusion, internal, construct and external threats to validity of our empirical evaluation.

Chapter 5

Summary and Conclusions

5.1 Summary

Mutation testing is a resource intensive process, potentially producing thousands of mutants for a given software system. Recall that mutation testing generates a set of mutants from the source code of the SUT and than evaluates these using the provided test suite. A mutation score is calculated as a result of mutation testing, which indicates how effective the test suite is at finding faults (i.e., test suite effectiveness).

There have been a number of research studies aimed at improving mutation testing performance by adjusting the mutation testing process (i.e., better mutation representation/generation/evaluation), instead we applied machine learning to predict the mutation score of source code units. As described in Chapter 3 we use a SVM to make predictions based on the features of class- and method-level source code units. We use the source code and test suite as the source of metrics for our predictions as they are directly involved in the mutation testing process. Specifically, we identify four initial sets of metrics (i.e., feature sets) from the SUT: source code, coverage, accumulated source code, and accumulated test suite.

We evaluated our approach using eight test subjects that contained 1689 classes, 113686 method and 10233 test cases. Using the available data, we proposed five research questions in Section 4.3. The following summarizes our findings:

- In Section 4.3.1 we determined that the eight test subjects had effective test suites, all but one exceeding a 73% mutation score. By considering the distribution of mutation scores for our test subjects, we were able to determine three suitable ranges for mutation score categories to abstract the real-values of mutation scores.

- In Section 4.3.2 we used the available data and evaluated the different feature sets over all our test subjects. Using all feature sets provided the greatest cross-validation accuracy, outperforming the feature sets individually.

- In Section 4.3.3 we evaluated our prediction approach on unknown data within individual test subjects and across test subjects. Our results showed that class-level mutation score was more difficult to predict, while method-level predictions performed better than class-level predictions. Furthermore, method-level predictions of unknown data within an individual test subject yielded higher accuracy than across test subjects.

- In Section 4.3.4 we explored avenues to optimize and generalize our prediction approach. We identified that using cross-validation accuracy for SVM parameter selection can be ineffective. We presented several alternative and more effective performance measures, namely F-score. We performed our own grid search to identify a single set of SVM parameters that maximized F-score across all data sets. We identified generalizable SVM parameters for class- and method-level predictions, and re-evaluated our prediction accuracy using the new parameters.

As a result we were able to increase the average prediction accuracy for class- and method-level source code units by +5.0% and +3.6% within systems, and +4.6% and +5.2% across systems for class- and method-level source code units.

- In Section 4.3.5 we conducted an experiment that observed how our approach's prediction accuracy changed with the availability of data. By limiting the amount of training data used for training we showed that a learning curve was clearly defined in some of our test subjects while not as pronounced in others. This experiment demonstrates that it is possible to train on a fraction of the available data and predict the remainder with near optimal accuracy. We showed the applicability of using our approach for an iterative development environment where it is possible to use mutation testing on a fraction of the available data and predict on the remaining amount.

5.2 Contributions

Based on our empirical evaluation (see Chapter 4) we produced the following contributions to the domain of mutation score prediction:

- Proposed a new approach for predicting the mutation score of a class- and method-level source code unit using source code and test suite metrics. We performed an empirical evaluation of our approach over eight open source test subjects.

- Identified 33 specific metrics (further grouped into four logical sets) that characterize source code units with respect to mutation score predictions.

- Demonstrated through cross-validation that the four feature sets in combination provided more accuracy than that of the feature sets individually.

- Demonstrated that prediction on unknown data is possible. Predictions within an individual software system has more variation in the mutation score performance than predictions on an unknown software system. Specifically, we showed that using all the available features we achieved an average prediction accuracy within systems of 36.7% and 56.8%, for class- and method-level source code units. We also achieved an average prediction accuracy across systems of 39.0% and 42.8% for class- and method-level source code units.

- Identified a generalizable set of SVM parameters that maximized F-score over our test subjects. These parameters increased prediction accuracy (+5.0% for class-level within systems, +4.6% for class-level across systems, +3.6% for method-level within systems, and +5.2% for method-level across systems) over that was determined through cross-validation. We achieved higher than random prediction accuracies using generalized SVM parameters, which removes the need of finding suitable parameters for new data.

- Demonstrated that it is not necessary to train on 90% of the available data (as in 10-fold cross-validation) in order achieve near optimal prediction accuracy.

In summary, our approach is novel in that we considered both source code and test suite metrics as factors to make mutation score predictions. We also performed feature selection on the collected features in Appendix B, results indicated that it is possible to reduce the feature set and retain prediction accuracy.

5.3 Limitations

Several limitations of our approach and empirical evaluation include:

- In our approach we removed abstract, overloaded and anonymous source code unit (due to their complexity), which reduced our usable data and could misrepresented the actual test subject's software systems.

- We calculated the *NOT* (i.e., number of tests) metrics in our approach with *Javalanche*, ideally this should be calculated using *EMMA*.

- We used only eight open source test subjects, this is most likely not a representative sample of all software systems.

- We used only considered 33 possible source code and test suite metrics to predict mutation testing scores, mostly likely there are more metrics that could have been used for our predictions.

- We evaluated our approach using three categories to abstract the exact mutation score prediction. The results may not represent prediction on a finer grain set of mutation score categories.

Furthermore the issues concerning the threats to validity (see Section 4.4) are also limitations to our approach and empirical evaluation.

5.4 Future Work

The future work for this thesis can be divided into two topics:

- Improvement and optimization of our approach for predicting mutation scores (Section 5.4.1.

- Further experimentation with additional statistical measures to validate the generalizability of our results (Section 5.4.2).

Furthermore, obviously we would like to address the limitations listed in the previous section.

5.4.1 Optimizing and Improving Approach

In our approach we discard abstract/overloaded/anonymous source code units as they were a bit more complex to handle during the construction of our approach. As for future work it would be wise to reconsider these omitted details as they obviously contribute to the data (i.e., mainly for class-level source code units, which might explain why their prediction accuracy was lower). We would like to correctly determine the *NOT* (i.e., number of tests) metric without relying on *Javalanche* for coverage data. *EMMA* is fully capable in determining the *NOT* feature.

We would like to further explore additional metrics and other facets of a software system. For example, we would like to use the *Software Testing and Reliability Early Warning (STREW) metric suite* [NWO+05, NWVO05] or even the number of assertions within test cases. Furthermore, we could explore the mutants themselves as the mutation generation is not the expensive aspect of mutation testing, or even runtime information. Negappan et al. mined metrics to predict component failures and stated that *"Predictors are accurate only when obtained from the same or similar projects"* [NBZ06]. This suggests that prediction across software systems, as we did in for our empirical evaluation is not ideal. We would like to further investigate this with respect to our approach, and see if we can achieve higher prediction performance using similar projects while predicting across software systems.

By using *Javalanche* we unfortunately did not have access to class-level object-oriented mutation operators, and a limited subset of traditional method-level mutation operators. Future work would be required to add these missing mutation operators into *Javalanche* as it would not only benefit this thesis but also those that use *Javalanche*.

Furthermore, there exists more than the traditional mutation operators that generate typical faults, it would be a great addition to incorporate security and concurrency mutation operators into our approach. Equivalent mutants pose a challenge in interpreting mutation testing results, *Javalanche* has an approach that attempts to mitigate the impact of equivalent mutants that we ignored. Further work can integrate this consideration into our approach, we initially did not include it as it would further reduce our available data.

There are standard optimizations that can be done for our implementation such as better data structures and taking advantage of concurrency. We also would like to adapt our approach so that others can use it from a usability point-of-view. For example, a simple script that allows a user to specify source code unit(s) to be predicted based on a already trained classification model or as an Eclipse plugin. As our approach uses a classification approach for prediction, it is possible to extract from the SVM the probability that a vector belongs to a specific category. We would like to take advantage of this and present this data as well as it illustrates the confidence in the predictions.

5.4.2 Statistical and Experimental Evaluation

With our experimental setup we utilize a minimum of ten executions to reduce the noise in our results. To further reinforce our results, future work would involve increasing the number of executions (i.e., between 25 and 100 executions). In our analysis of the results we primarily used summary statistics (i.e., mean, standard deviation, etc...), while these statistics provide valid summary of the results there are other statistical measures that are stronger (i.e., confidence interval, hypothesis testing, etc...). Furthermore we performed a number of experiments to evaluate our approach, these alone are not comprehensive in what could have been done. Additional

analysis on the features and their relationship with each other and mutation scores would be an interesting study to conduct as it may provide additional detail on the source code units. Further investigation on the data is required to understand whether the predictive ability of our approach depends on the distribution/availably of data and/or the features used. There still remains a lot of experimentation to be done in this area with respect to our approach. For future work we would like to evaluate our current results and new experiments with stronger statistical measures.

With respect to our implementation we utilized a number of tools to gather features and the mutation scores. We would like to explore using other tools as alternative as this can show that our approach still functions correctly independently of the tools used. Due to our limited set of test subjects we did not have a wide variety of domains, source sizes, test suite sizes and mutation scores. By including more open source and potentially industrial software systems we can cover more pairings of the aforementioned criteria, which will shine insight on the generalizability of our approach.

5.5 Conclusions

Mutation testing is just too costly, which inhibits industry adoption. We stated the following in Chapter 1:

> ***Thesis Statement:*** *The use of source code and test suite metrics in combination with machine learning techniques can accurately predict mutation scores. Furthermore, the predictions can be used to reduce the performance cost of mutation testing when used to iteratively develop test suites.*

We followed through with the thesis statement with the creation of such an approach to predict mutation scores using source code and test suite metrics. We

discussed the necessary topics required for this thesis in Chapter 2. We described our approach along-side an example in Chapter 3. We outlined a set of experiments in Chapter 4 that evaluated our approach to answer several research questions related to our thesis statement. Finally we present limitations, threats to validity and future work in Chapter 5.

With our approach we showed that it is indeed possible to predict mutation scores of source code units using source code and test suite metrics. For predictions on unknown source code units within a software system, we were able to achieve an average accuracy of 56.8% for method-level predictions. Exploratory work on method-level prediction of unknown data across software systems provided lesser accuracy at 42.8%. Both of these values are higher than random prediction accuracy (i.e., 33.3%) using a general set of SVM parameters, which eases the complexity of tuning our technique. Class-level predictions did not fare as well compared to method-level predictions, with 36.7% accuracy within systems and 39.0% accuracy across systems. Contrary to other prediction techniques (i.e., bug detection) we observed the test suite in addition to the source code, which is quite novel. With future work we hope that test suite metrics can be further used in existing and future research. Furthermore we anticipate that our approach still has room for improvement with respect to generalizability and prediction accuracy.

Bibliography

[ABD+79] A.T. Acree, T.A. Budd, R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Mutation analysis. Technical Report GIT-ICS-79/08, School of Information and Computer Science, Georgia Institute of Technology, Sep. 1979.

[ABL05] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proc. of the 27th Int Conf. on Soft. Eng. (ICSE '05)*, pages 402–411, May 2005.

[ABLN06] J.H. Andrews, L.C. Briand, Y. Labiche, and A.S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Trans. on Soft. Eng.*, 32(8):608–624, Aug. 2006.

[ABM98] P.E Ammann, P.E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proc. of the 2nd IEEE Int. Conf. on Formal Eng. Methods (ICFEM'98)*, pages 46–54, Dec. 1998.

[AC94] F.B. Abreu and R. Carapuça. Object-oriented software engineering: Measuring and controlling the development process. In *Proc. of the 4th Int. Conf. on Soft. Quality*, 1994.

[AD91] H. Almuallim and T.G. Dietterich. Learning with many irrelevant features. In *Proceedings of the 9th National Conf. on Artificial intelligence (AAAI '91)*, volume 2, pages 547–552, 1991.

[AE09] R Abraham and M Erwig. Mutation operators for spreadsheets. *IEEE Trans. on Soft. Eng.*, 35(1):94–108, Jan.–Feb. 2009.

[AKJ04] R. Akbani, S. Kwek, and N. Japkowicz. Applying support vector machines to imbalanced datasets. *Proc. of the 11th European Conf. on Machine Learning (ECML '04)*, pages 39–50, 2004.

[Alp04] E. Alpaydin. *Introduction to Machine Learning.* The MIT Press, 2004.

[AMM95] C. Anderson, A.von Mayrhauser, and R.T. Mraz. On the use of neural networks to guide software testing activities. In *Proc. of the IEEE Int. Test Conf.*, pages 720–729, 1995.

[bar] Barbecue. web page: http://barbecue.sourceforge.net/, (last accessed Jun. 11, 2012).

[BCD06] J.S. Bradbury, J.R. Cordy, and J. Dingel. Mutation operators for concurrent Java (J2SE 5.0). In *Proc. of the 2nd Work. on Mutation Analysis (Mutation 2006)*, pages 83–92, Nov. 2006.

[BDLS80] T. A. Budd, R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *Proc. of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '80*, pages 220–233, 1980.

[Bec] K. Beck. JUnit. web page: http://www.junit.org/, (last accessed May 18, 2012).

[BHW10] A. Ben-Hur and J. Weston. A user's guide to support vector machines. *Methods in Molecular Biology*, 609:223–239, 2010.

[BL97] A.L. Blum and P. Langley. Selection of relevant features and examples in machine learning. *Artificial intelligence*, 97(1-2):245–271, 1997.

[BL07] J. Bennett and S. Lanning. The Netflix prize. In *Proceedings of KDD Cup and Workshop*, volume 2007, pages 3–6, 2007.

[BOSB10] K.H. Brodersen, C.S. Ong, K.E. Stephan, and J.M. Buhmann. The balanced accuracy and its posterior distribution. In *Proc. of the 2010 20th Int. Conf. on Pattern Recognition (ICPR '10)*, pages 3121–3124, 2010.

[BPM04] G.E. Batista, R.C. Prati, and M.C. Monard. A study of the behavior of several methods for balancing machine learning training data. *ACM SIGKDD Explorations Newsletter*, 6(1):20–29, Jun. 2004.

[Bud80] T.A. Budd. *Mutation Analysis of Program Test Data*. Ph.D. Thesis, Yale University, 1980.

[BVSF04] R. Barandela, R. Valdovinos, J. Sanchez, and F. Ferri. The imbalanced training sample problem: Under or over sampling? *Structural, Syntactic, and Statistical Pattern Recognition*, pages 806–814, 2004.

[CK94] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. on Soft. Eng.*, 20(6):476–493, Jun 1994.

[CL11] C.C. Chang and C.J. Lin. LIBSVM: A library for support vector machines. *ACM Trans. Intell. Syst. Technol. (TIST)*, 2:27:1–27:27, May 2011. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[Col] H. Coles. PIT mutation testing. web page: `http://pitest.org/`, (last accessed May 18, 2012).

[com] Apache Commons Lang. web page: `http://commons.apache.org/lang/`, (last accessed Jun. 11, 2012).

[CV95] C. Cortes and V. Vapnik. Support-vector networks. *Mach. Learn.*, 20(3):273–297, Sept. 1995.

[DLS78] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, Apr. 1978.

[DM96] M.E. Delamaro and J.C. Maldonado. Proteum – a tool for the assessment of test adequacy for C programs. In *Proc. of the Conf. on Performability in Computing Sys. (PCS'96)*, number SERC-TR-168-P, pages 79–95, 1996.

[FB99] M. Fowler and K. Beck. *Refactoring: improving the design of existing code.* Addison-Wesley Longman Publishing Co., Inc., 1999.

[Fen94] N. Fenton. Software measurement: a necessary scientific basis. *IEEE Trans. on Soft. Eng.*, 20(3):199–206, Mar. 1994.

[FP98] N.E. Fenton and S.L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach.* International Thomson Computer Press, 2nd edition, 1998.

[GE03] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *The J. of Machine Learning Research*, 3:1157–1182, 2003.

[GFS05] T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Trans. of Soft. Eng.*, 31(10):897–910, Oct. 2005.

[GJ08] A. Gupta and P. Jalote. An approach for experimentally evaluating effectiveness and efficiency of coverage criteria for software testing. *Int. J. Soft. Tools Technology. Transfer (STTT).*, 10(2):145–160, Feb. 2008.

[Goo93] P. Goodman. *The Practical Implementation of Software Metrics.* McGraw-Hill, 1993.

[Gun98] S.R. Gunn. Support vector machines for classification and regression. Technical report, University of Southampton, 1998.

[Hal99] M.A. Hall. *Correlation-based feature selection for machine learning.* PhD thesis, The University of Waikato, 1999.

[HCL03] C.W. Hsu, C.C. Chang, and C.J. Lin. A practical guide to support vector classification. Technical report, National Taiwan University, 2003.

[Her00] J. Herbst. A machine learning approach to workflow management. *Proc. of the 11th European Conf. on Machine Learning (ECML '00)*, pages 183–194, 2000.

[Her03] A. Hertzmann. Machine learning for computer graphics: A manifesto and tutorial. In *Proc. of the 11th Pacific Conf. on Computer Graphics and Applications*, pages 22–36, 2003.

[HFH+09] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I.H. Witten. The WEKA data mining software: an update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, Nov. 2009.

[HS96] B. Henderson-Sellers. *Object-oriented metrics: measures of complexity.* Prentice-Hall, Inc., 1996.

[HWY09] T. Honglei, S. Wei, and Z. Yanan. The research on software metrics and software complexity metrics. In *Proc. of Int. Forum on Computer Science-Technology and Applications (IFCSTA '09)*, pages 131–136, Dec. 2009.

[Ino12] L.M.M.L. Inozemtseva. Predicting test suite effectiveness for Java programs. Master's thesis, University of Waterloo, 2012.

[Jap00] N. Japkowicz. The class imbalance problem: Significance and strategies. In *Proc. of the 2000 Int. Conf. on Artificial Intelligence (ICAI '00)*, volume 1, pages 111–117, 2000.

[JB12] K. Jalbert and J.S. Bradbury. Predicting mutation score using source code and test suite metrics. In *Proc. of the Work. on Realizing Artificial Intelligence Synergies in Soft. Eng. (RAISE 2012)*, Jun. 2012.

[jga] JGAP – Java genetic algorithms and genetic programming package. web page: `http://jgap.sourceforge.net/`, (last accessed Jun. 11, 2012).

[JH08] Y. Jia and M. Harman. MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language. In *Testing: Academic & Industrial Conf. – Practice and Research Techniques (TAIC PART 2008)*, pages 94–98, 2008.

[JH11] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. on Soft. Eng.*, 37(5):649–678, Sep.–Oct. 2011.

[JKP94] G.H. John, R. Kohavi, and K. Pfleger. Irrelevant features and the subset selection problem. In *Proc. of the 11th Int. Conf. on Machine Learning*, volume 129, pages 121–129, 1994.

[Joa99] T. Joachims. Advances in kernel methods. chapter Making large-scale support vector machine learning practical, pages 169–184. 1999.

[joda] Joda-Primitives. web page: `http://joda-primitives.sourceforge.net/`, (last accessed Jun. 11, 2012).

[jodb] Joda-Time. web page: `http://joda-time.sourceforge.net/`, (last accessed Jun. 11, 2012).

[jso] jsoup. web page: `http://jsoup.org/`, (last accessed Jun. 11, 2012).

[Jum] Jumble. web page: `http://jumble.sourceforge.net/`, (last accessed May 18, 2012).

[Kan02] S.H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 2002.

[KJ97] R. Kohavi and G.H. John. Wrappers for feature subset selection. *Artificial intelligence*, 97(1-2):273–324, 1997.

[KL05] A.G. Koru and H. Liu. Building effective defect-prediction models in practice. *IEEE Soft.*, 22(6):23–29, Nov.–Dec. 2005.

[KM09] B. Kitchenham and E. Mendes. Why comparative effort prediction studies may be invalid. In *Proc. of the 5th Int. Conf. on Predictor Models in Soft. Eng. (PROMISE '09)*, pages 4:1–4:5, 2009.

[KO91] K.N. King and A.J. Offutt. A Fortran language system for mutation-based software testing. *Software: Practice and Experience*, 21(7):685–718, 1991.

[Koh95] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proc. of the 14th Int. Joint Conf. on Artificial Intelligence (IJCAI '95)*, pages 1137–1143, 1995.

[Kon01] I. Kononenko. Machine learning for medical diagnosis: history, state of the art and perspective. *Artificial Intelligence in medicine*, 23(1):89–109, 2001.

[KR92] K. Kira and L.A. Rendell. The feature selection problem: Traditional methods and a new algorithm. In *Proceedings of the 10th National Conf. on Artificial Intelligence (AAAI '92)*, pages 129–129, 1992.

[log] Logback. web page: `http://logback.qos.ch/`, (last accessed Jun. 11, 2012).

[McC76] T.J. McCabe. A complexity measure. *IEEE Trans. on Soft. Eng.*, 2(4):308–320, Dec. 1976.

[Met] Eclipse Metrics plugin. web page: `http://metrics2.sourceforge.net/`, (last accessed Mar. 1, 2012).

[MKO02] Y.S. Ma, Y.R Kwon, and J. Offutt. Inter-class mutation operators for Java. In *Proc. of the 13th Int. Symp. on Soft. Reliability Eng. (ISSRE 2002)*, pages 352–363, 2002.

[MKPS00] S. Muthanna, Kontogiannis K., K. Ponnambalam, and B. Stacey. A maintainability model for industrial software systems using design level metrics. In *Proc. of the 7th Working Conf. on Reverse Eng. (WCRE 2000)*, pages 248–256, 2000.

[MO05a] Y.S. Ma and J. Offutt. Description of class mutation mutation operators for Java. web page: `http://cs.gmu.edu/~offutt/mujava/mutopsClass.pdf`, (last accessed Jun. 11, 2012), Nov. 2005.

[MO05b] Y.S. Ma and J. Offutt. Description of method-level mutation operators for Java. web page: `http://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf`, (last accessed Jun. 11, 2012), Nov. 2005.

[MOK05] Y.S. Ma, J. Offutt, and Y.R. Kwon. MuJava: an automated class mutation system. *Software Testing, Verification and Reliability*, 15(2):97–133, 2005.

[Moo] I. Moore. Jester. web page: `http://jester.sourceforge.net/`, (last accessed May 18, 2012).

[MR10] L. Madeyski and N. Radyk. Judy – a mutation testing tool for Java. *IET Software*, 4(1):32–42, Feb. 2010.

[NA09] A.S. Namin and J.H. Andrews. The influence of size and coverage on test suite effectiveness. In *Proc. of the 18th Int. Symposium on Soft. Testing and Analysis (ISSTA '09)*, pages 57–68, 2009.

[NBP08] G.H. Nguyen, A. Bouzerdoum, and S.L. Phung. Efficient supervised learning with reduced training exemplars. In *IEEE 2008 Int. Joint Conf. on Neural Networks (IJCNN '08)*, pages 2981–2987, Jun. 2008.

[NBZ06] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering (ICSE '06)*, pages 452–461, 2006.

[NK11] A.S. Namin and S. Kakarla. The use of mutation in testing experiments and its sensitivity to external threats. In *Proc. of the 2011 Int. Symposium on Soft. Testing and Analysis (ISSTA '11)*, pages 342–352, 2011.

[NWO+05] N. Nagappan, L. Williams, J. Osborne, M. Vouk, and P. Abrahamsson. Providing test quality feedback using static source code and automatic test suite metrics. In *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering (ISSRE '05)*, pages 85–94, 2005.

[NWVO05] N. Nagappan, L. Williams, M. Vouk, and J. Osborne. Early estimation of software quality using in-process testing metrics: a controlled case study. In *Proceedings of the third workshop on Software quality*, pages 1–7, 2005.

[OAL06] J. Offutt, P. Ammann, and L. Liu. Mutation testing implements grammar-based testing. In *Proc. of the 2nd Work. on Mutation Analysis (Mutation 2006)*, page 12, Nov. 2006.

[OC94] A.J. Offutt and W.M. Craft. Using compiler optimization techniques to detect equivalent mutants. *Software Testing, Verification and Reliability*, 4(3):131–154, 1994.

[Off92] A.J. Offutt. Investigations of the software testing coupling effect. *ACM Trans. on Soft. Eng. and Methodology (TOSEM)*, 1(1):5–20, Jan. 1992.

[OLP08] J.D. Olden, J.J. Lawler, and N.L.R. Poff. Machine learning methods without tears: a primer for ecologists. *The Quarterly review of biology*, 83(2):171–193, 2008.

[OLR+96] A.J. Offutt, A. Lee, G. Rothermel, R.H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Soft. Eng. Methodol.*, 5(2):99–118, Apr. 1996.

[OMK04] A.J. Offutt, Y.S. Ma, and Y.R. Kwon. An experimental mutation system for Java. *ACM SIGSOFT Soft. Eng. Notes*, 29(5):1–4, Sep. 2004.

[ope] OpenFAST. web page: `http://www.openfast.org/`, (last accessed Jun. 11, 2012).

[OU01] A.J. Offutt and R.H. Untch. Mutation 2000: uniting the orthogonal. In *Mutation testing for the new century*, pages 34–44, 2001.

[PJO99] F. Provost, D. Jensen, and T. Oates. Efficient progressive sampling. In *Proc. of the 5th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD '99)*, pages 23–32, 1999.

[PO10] U. Praphamontripong and J. Offutt. Applying mutation testing to web applications. In *Proc. of the 2010 3rd Int. Conf. on Soft. Testing, Verification, and Validation Work. (ICSTW '10)*, pages 132–141, 2010.

[PSVG+02] K. Pelckmans, J.A.K. Suykens, T. Van Gestel, J. De Brabanter, L. Lukas, B. Hamers, B. De Moor, and J. Vandewalle. LS-SVMlab: a MATLAB/C toolbox for least squares support vector machines. Technical Report 02-44, ESAT-SISTA; K.U. Leuven, 2002.

[Res02] Research Triangle Institute. The economic impacts of inadequate infrastructure for software testing. planning report 02-3. Technical report, National Institute of Standards and Technology, United States, May 2002.

[Rou] V. Roubtsov. EMMA: A free Java code coverage tool. web page: `http://emma.sourceforge.net/`, (last accessed Mar. 1, 2012).

[SDZ09] D. Schuler, V. Dallmeier, and A. Zeller. Efficient mutation testing by checking invariant violations. In *Proc. of the 18th Int. Symposium on Soft. Testing and Analysis (ISSTA '09)*, pages 69–80, Jul. 2009.

[SJS06] M. Sokolova, N. Japkowicz, and S. Szpakowicz. Beyond accuracy, f-score and roc: a family of discriminant measures for performance evaluation. *AI 2006: Advances in Artificial Intelligence*, pages 1015–1021, 2006.

[SRD12] A. Shaik, C.R.K. Reddy, and A. Damodaram. Object oriented software metrics and quality assessment: Current state of the art. *Int. J. of Computer Applications*, 37(11):6–15, Jan. 2012.

[SRH+10] S. Sonnenburg, G. Rätsch, S. Henschel, C. Widmer, J. Behr, A. Zien, F. Bona, A. Binder, C. Gehl, and V. Franc. The SHOGUN machine learning toolbox. *J. of Machine Learning Research*, 99:1799–1802, Aug. 2010.

[SS08] P. Singh and H. Singh. DynaMetrics: a runtime metric-based analysis tool for object-oriented software systems. *SIGSOFT Soft. Eng. Notes*, 33(6):1–6, Nov. 2008.

[SV99] J.A.K. Suykens and J. Vandewalle. Least squares support vector machine classifiers. *Neural processing letters*, 9(3):293–300, 1999.

[SZ08a] H. Shahriar and M. Zulkernine. MUSIC: Mutation-based SQL injection vulnerability checking. In *Proc. of the 2008 The 8th Int. Conf. on Quality Soft.*, pages 77–86, Aug. 2008.

[SZ08b] H. Shahriar and M. Zulkernine. Mutation-based testing of buffer overflow vulnerabilities. In *Proc. of the 2008 32nd Annual IEEE Int. Computer Soft. and Applications Conf. (COMPSAC '08)*, pages 979–984, 2008.

[SZ09a] D. Schuler and A. Zeller. Javalanche: Efficient mutation testing for Java. In *Proc. of the the 7th Joint Meeting of the European Soft. Eng. Conf.*

and the ACM SIGSOFT Symposium on The Foundations of Soft. Eng. (ESEC/FSE '09), pages 297–298, 2009.

[SZ09b] H. Shahriar and M. Zulkernine. MUTEC: Mutation-based testing of cross site scripting. In *Proc. of the 2009 ICSE Work. on Soft. Eng. for Secure Systems (IWSESS '09)*, pages 47–53, 2009.

[SZ10] D. Schuler and A. Zeller. (Un-)covering equivalent mutants. In *Proc. of the 3rd Int. Conf. on Soft. Testing, Verification and Validation (ICST '10)*, pages 45–54, Apr. 2010.

[UOH93] R.H. Untch, A.J. Offutt, and M.J. Harrold. Mutation analysis using mutant schemata. *ACM SIGSOFT Soft. Eng. Notes*, 18(3):139–148, 1993.

[WDC10] W.E. Wong, V. Debroy, and B. Choi. A family of code coverage-based heuristics for effective fault localization. *J. Syst. Softw.*, 83(2):188–208, Feb. 2010.

[Wey93] E.J. Weyuker. Can we measure software testing effectiveness? In *Proc. of the 1st Int. Soft. Metrics Symposium*, pages 100–107, 1993.

[WFH11] I.H. Witten, E. Frank, and M.A. Hall. *Data Mining: Practical machine learning tools and techniques.* Morgan Kaufmann Publishers Inc., 3rd edition, 2011.

[WKP10] H.K. Wright, M. Kim, and D.E. Perry. Validity concerns in software engineering research. In *Proc. of the FSE/SDP Work. on Future of Soft. Eng. research (FoSER '10)*, pages 411–414, 2010.

[WNC05] J. Wang, P. Neskovic, and L.N. Cooper. Training data selection for support vector machines. In *Proc. of the 1st Int. Conf. on Advances in Natural Computation - Volume Part I (ICNC '05)*, pages 554–564, 2005.

[WRH+00] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering: an introduction.* Kluwer Academic Publishers, 2000.

[ZHM97] H. Zhu, P.A.V. Hall, and J.H.R. May. Software unit test coverage and adequacy. *ACM Computing Surveys (CSUR)*, 29(4):366–427, Dec. 1997.

Appendix A

Mutation Score Distributions

The individual mutation score distributions of class- and method-level source code units from the test subjects (see Section 4.3.1) are displayed within this appendix. The collective mutation score distribution of all test subjects is shown and described in Section 4.3.1. Statistical summary of each test subject's mutation score distribution is presented in Table A.1 and A.2.

Table A.1: Statistical summary of the class-level data for each test subject's mutation score.

	25^{th} Percentile	**50^{th} Percentile**	**75^{th} Percentile**	**Count**	**Min**	**Max**	**Sum**
logback-core	53%	74%	83%	100	0	7	115
barbecue	46%	70%	83%	100	0	3	31
jgap	53%	74%	87%	100	0	16	124
commons-lang	76%	81%	87%	100	0	12	124
joda-time	77%	86%	91%	100	0	17	194
openfast	74%	85%	93%	100	0	25	120
jsoup	77%	88%	94%	100	0	17	83
joda-primitives	77%	79%	85%	100	0	10	73
all	**72%**	**81%**	**89%**	**100**	**0**	**96**	**864**

Table A.2: Statistical summary of the method-level data for each test subject's mutation score.

	25^{th} Percentile	**50^{th} Percentile**	**75^{th} Percentile**	**Count**	**Min**	**Max**	**Sum**
logback-core	61%	83%	99%	100	0	145	447
barbecue	49%	77%	90%	100	0	33	143
jgap	59%	80%	99%	100	0	211	655
commons-lang	77%	87%	99%	100	0	288	789
joda-time	83%	94%	94%	100	0	974	2019
openfast	77%	87%	99%	100	0	159	401
jsoup	79%	91%	99%	100	0	166	381
joda-primitives	71%	83%	99%	100	0	207	675
all	**75%**	**87%**	**99%**	**100**	**0**	**2183**	**5510**

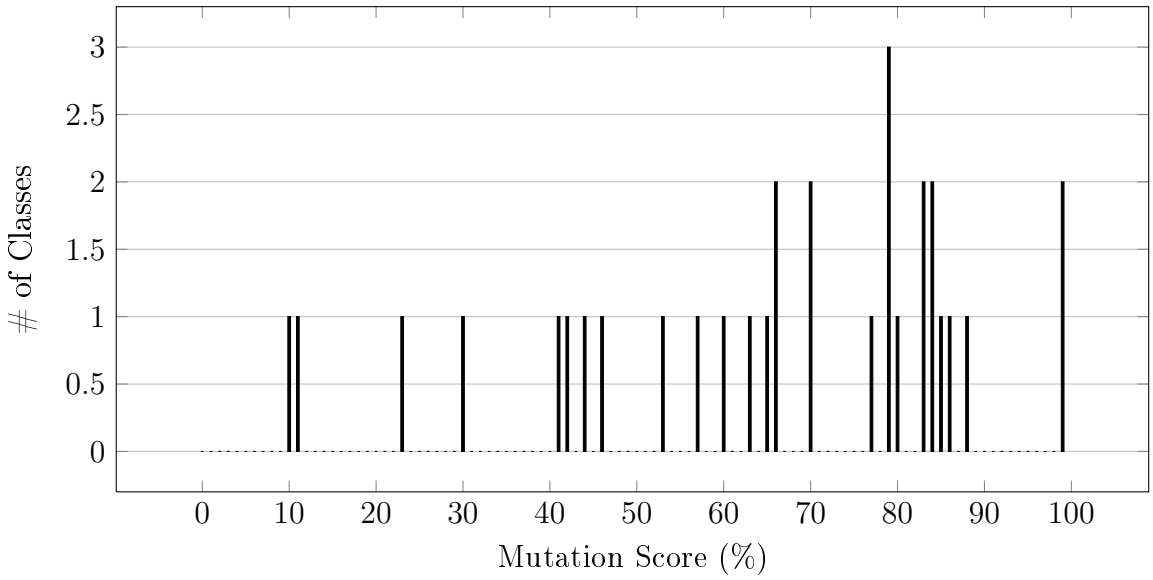


Figure A.1: Mutation score distribution of classes from *barbecue* that can be used for training.

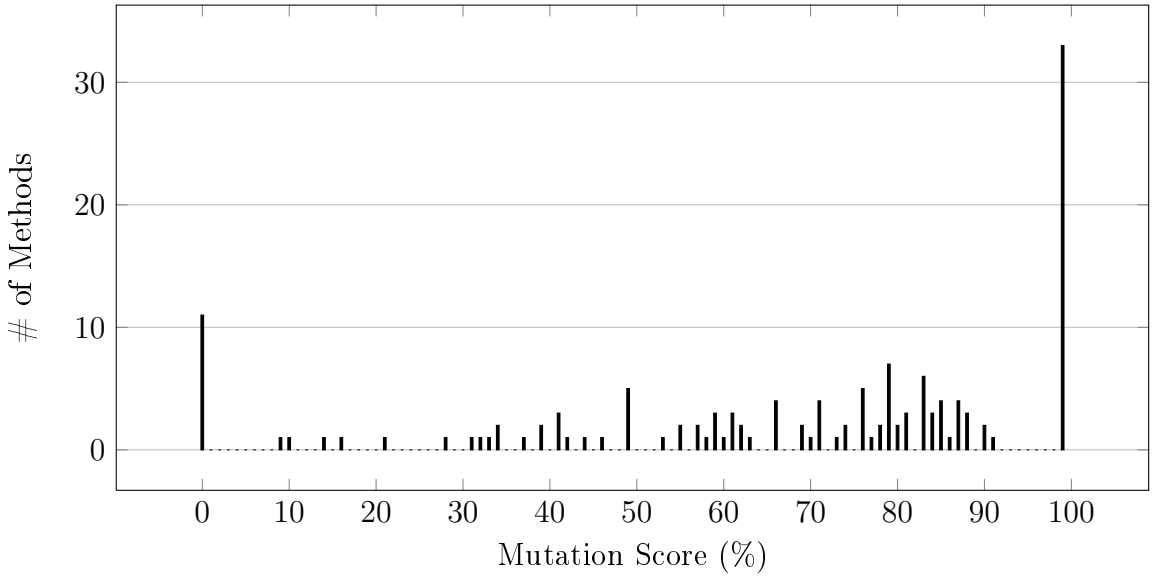


Figure A.2: Mutation score distribution of methods from *barbecue* that can be used for training.

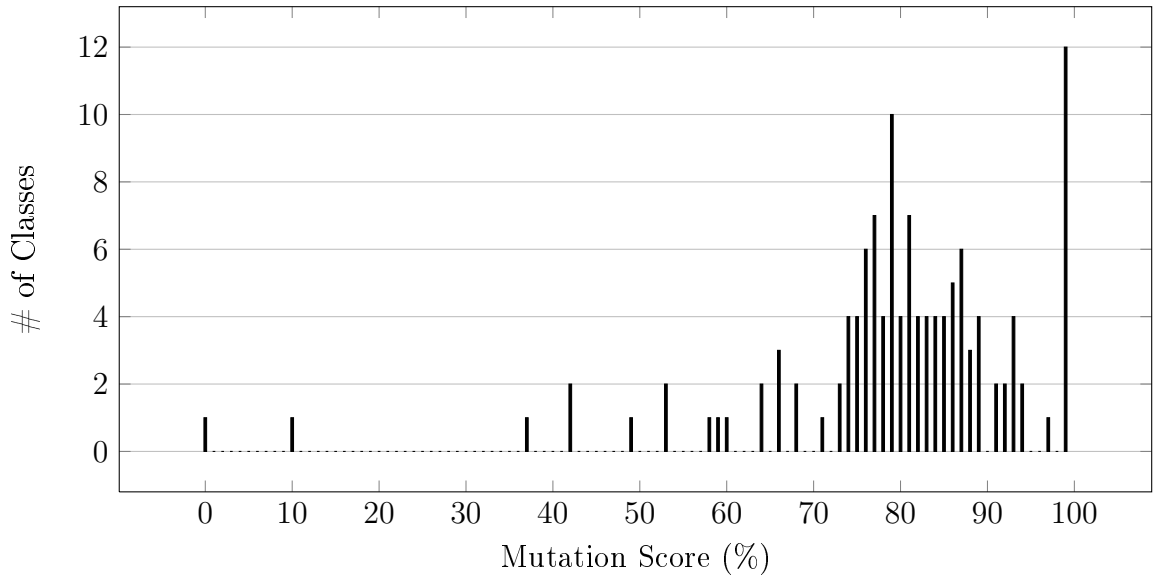


Figure A.3: Mutation score distribution of classes from *commons-lang* that can be used for training.

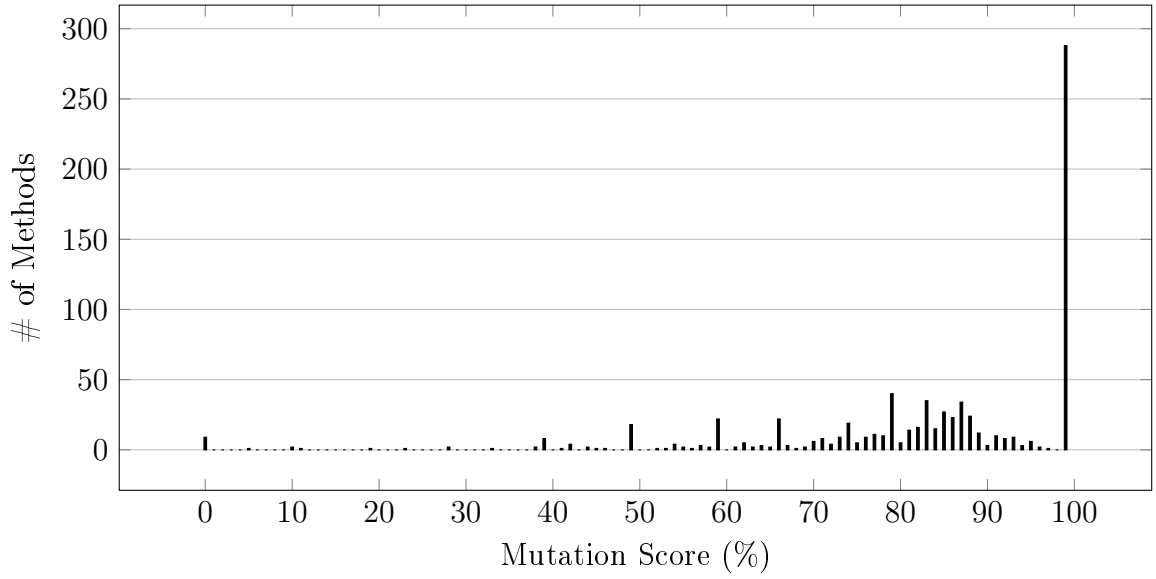


Figure A.4: Mutation score distribution of methods from *commons-lang* that can be used for training.

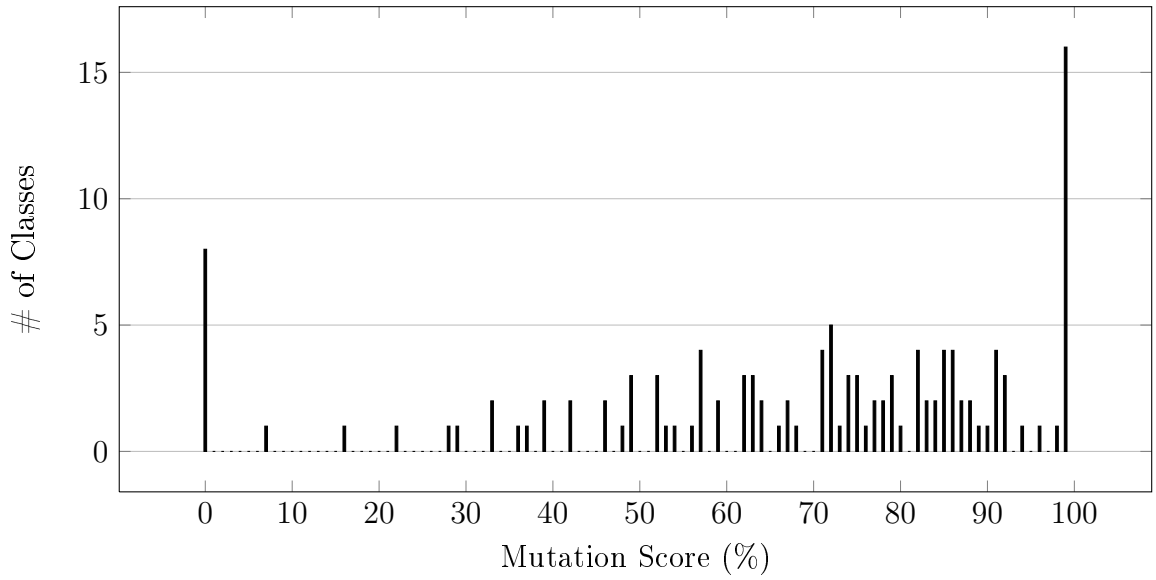


Figure A.5: Mutation score distribution of classes from *jgap* that can be used for training.

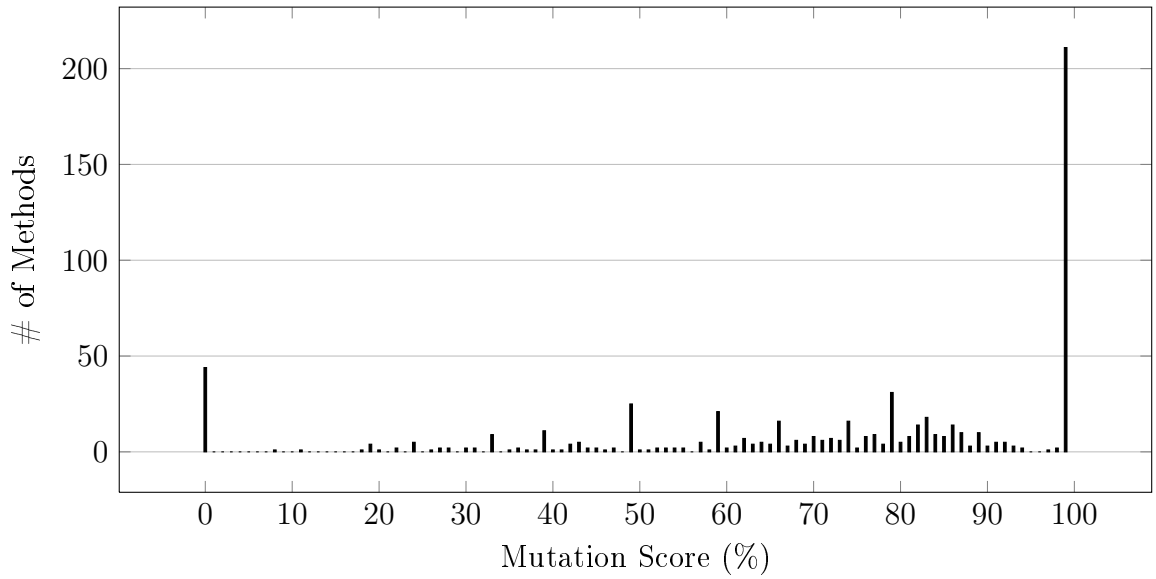


Figure A.6: Mutation score distribution of methods from *jgap* that can be used for training.

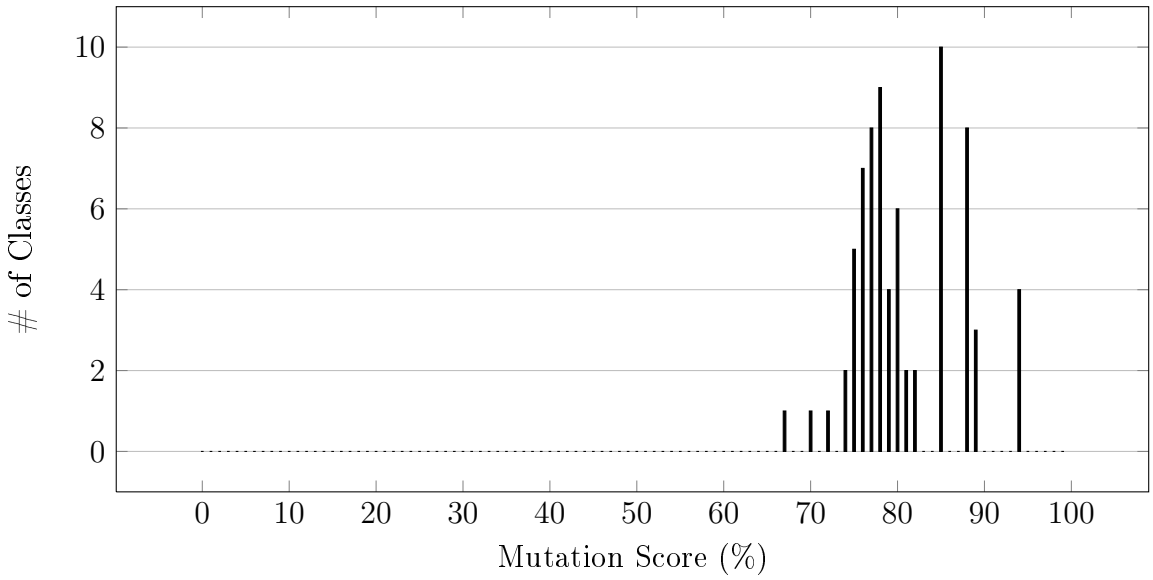


Figure A.7: Mutation score distribution of classes from *joda-primitives* that can be used for training.

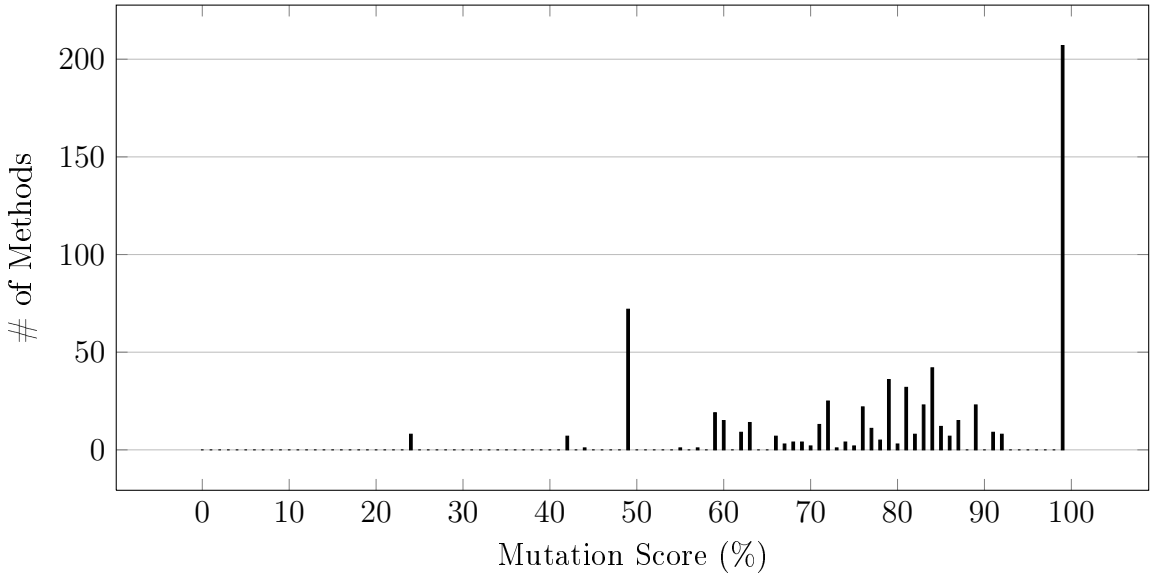


Figure A.8: Mutation score distribution of methods from *joda-primitives* that can be used for training.

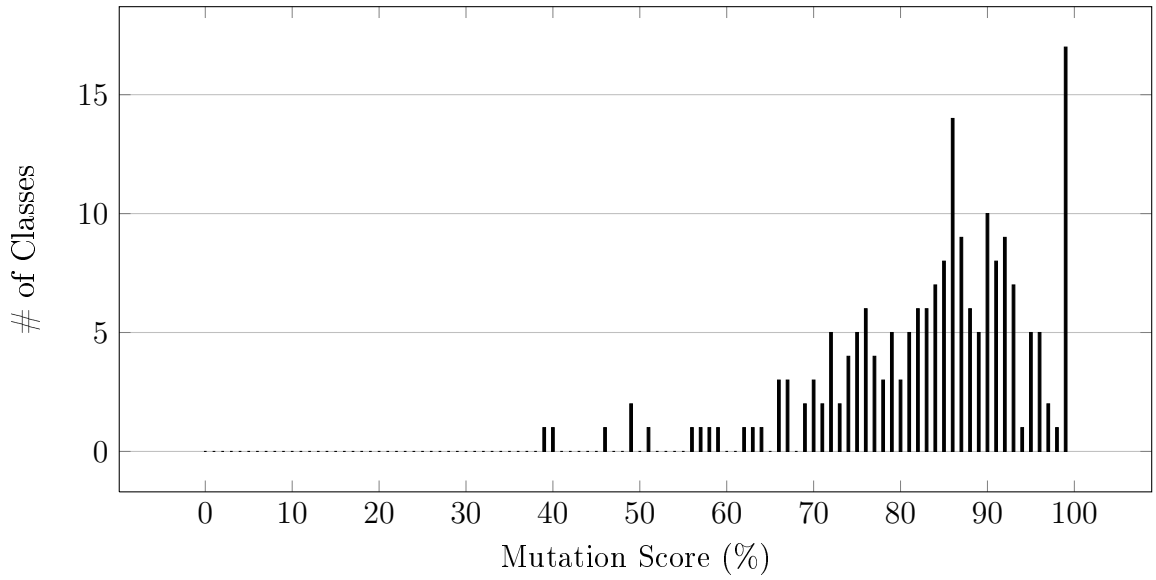


Figure A.9: Mutation score distribution of classes from *joda-time* that can be used for training.

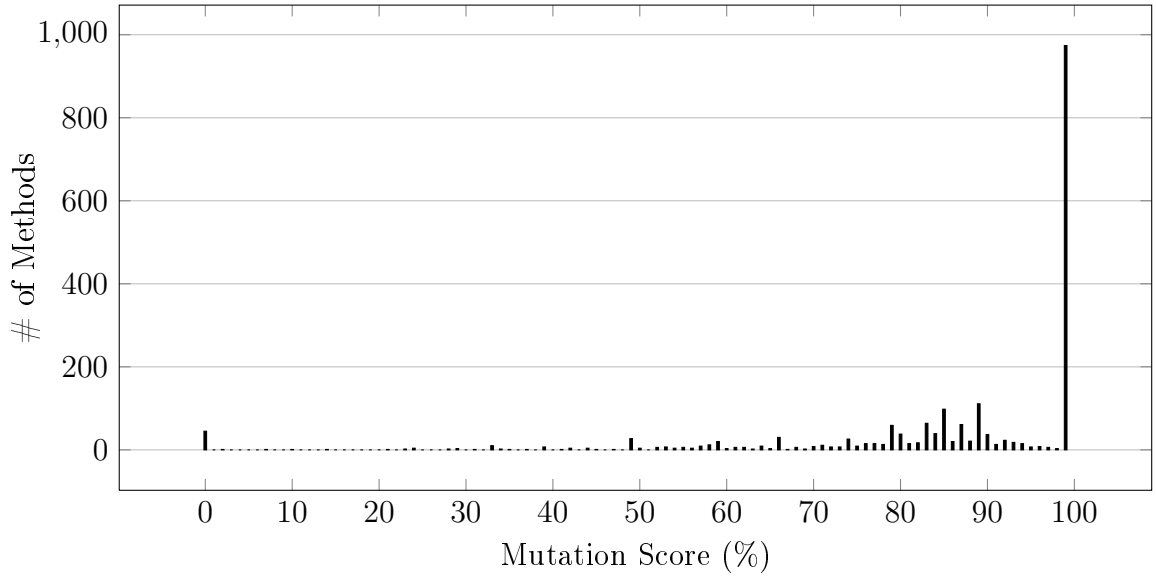


Figure A.10: Mutation score distribution of methods from *joda-time* that can be used for training.

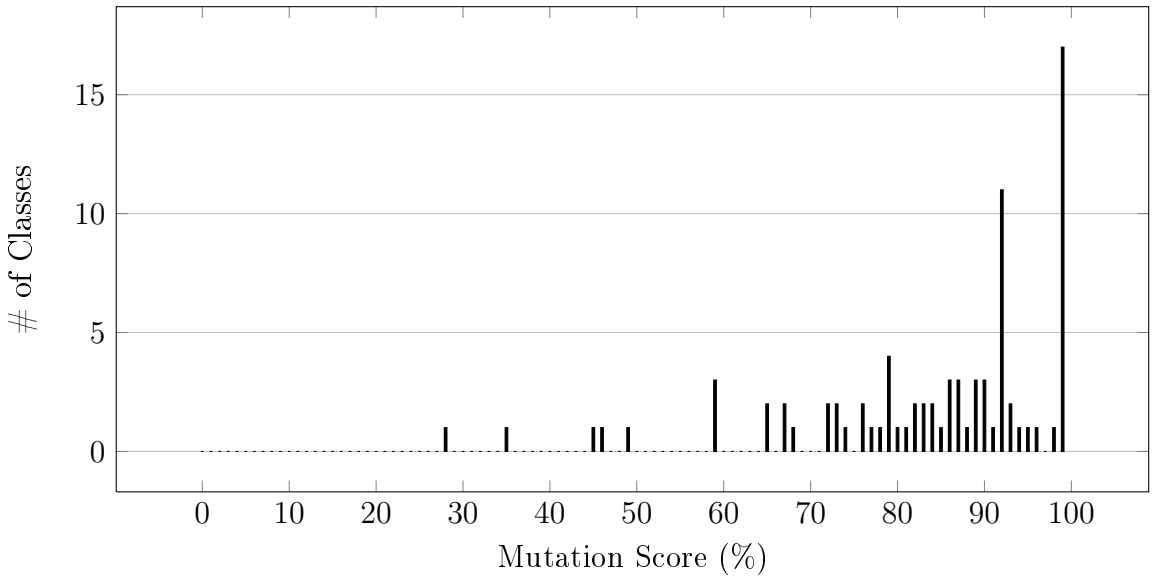


Figure A.11: Mutation score distribution of classes from *jsoup* that can be used for training.

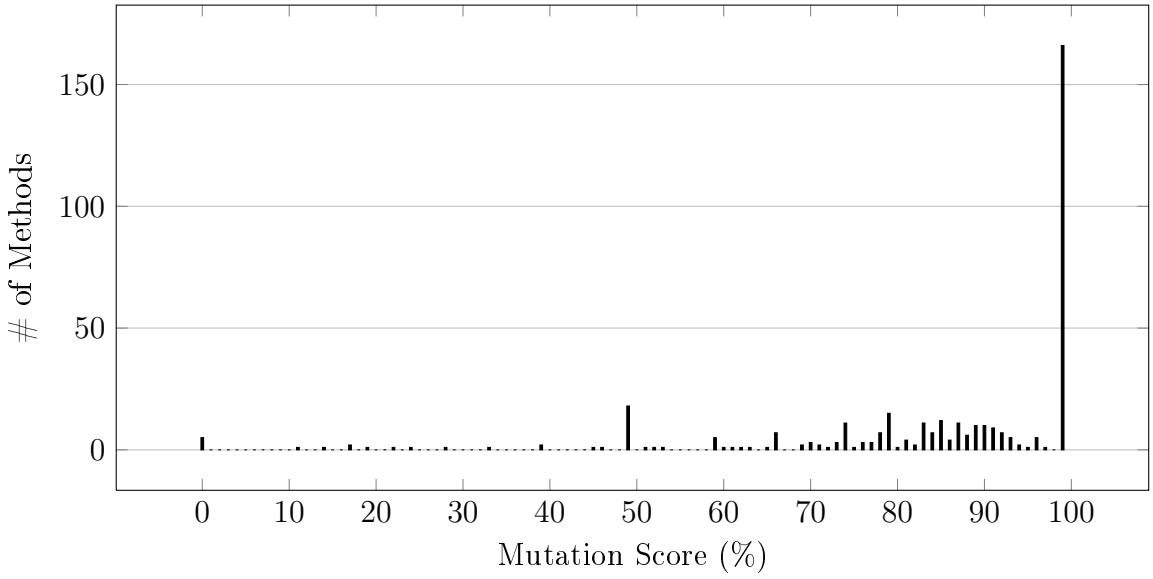


Figure A.12: Mutation score distribution of methods from *jsoup* that can be used for training.

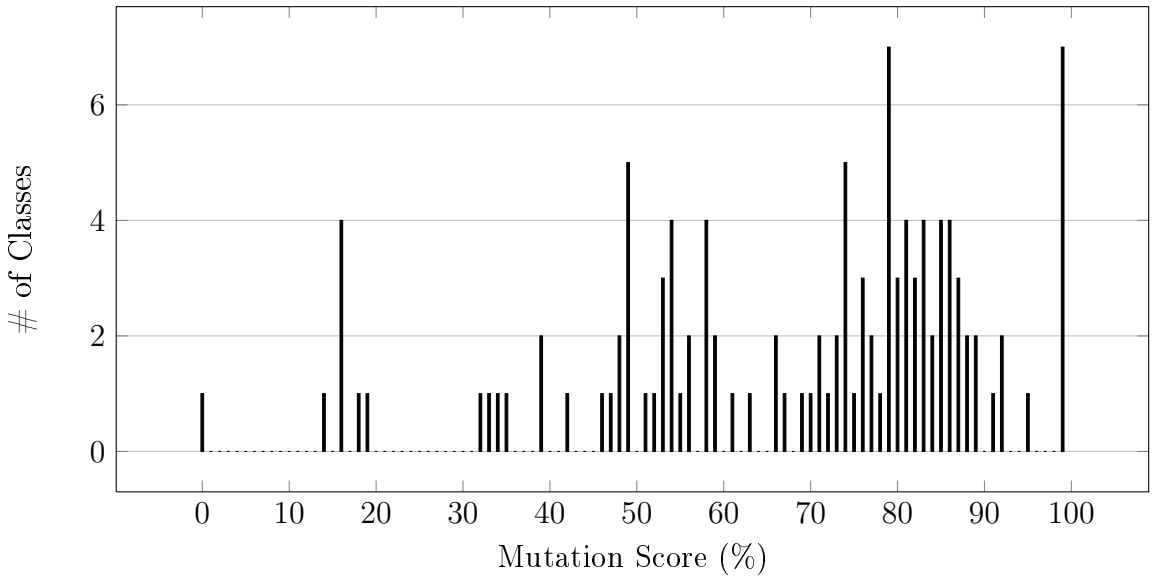


Figure A.13: Mutation score distribution of classes from *logback-core* that can be used for training.

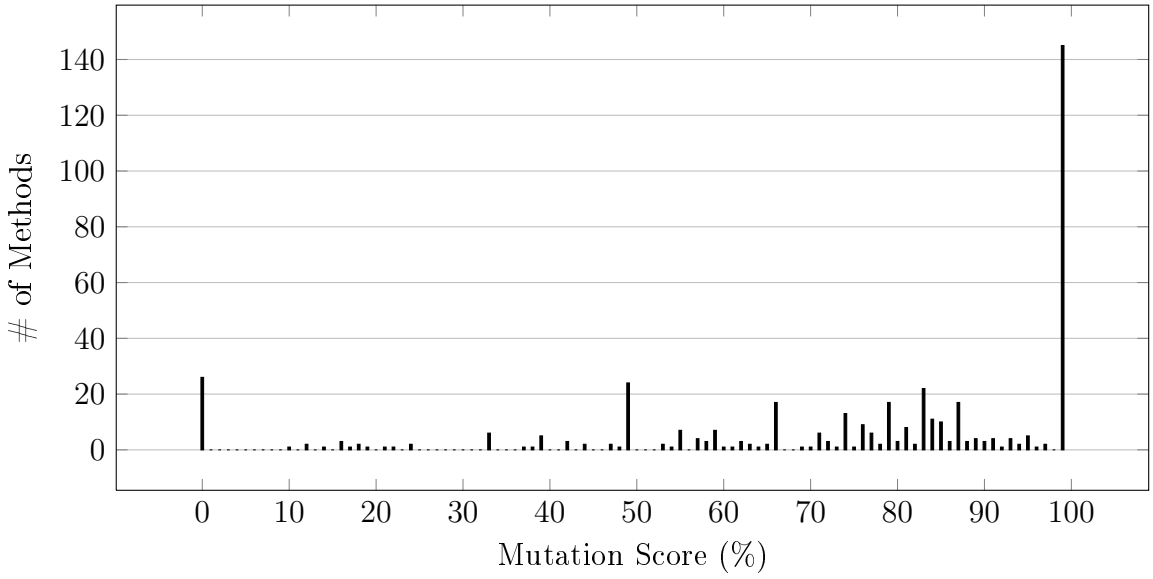


Figure A.14: Mutation score distribution of methods from *logback-core* that can be used for training.

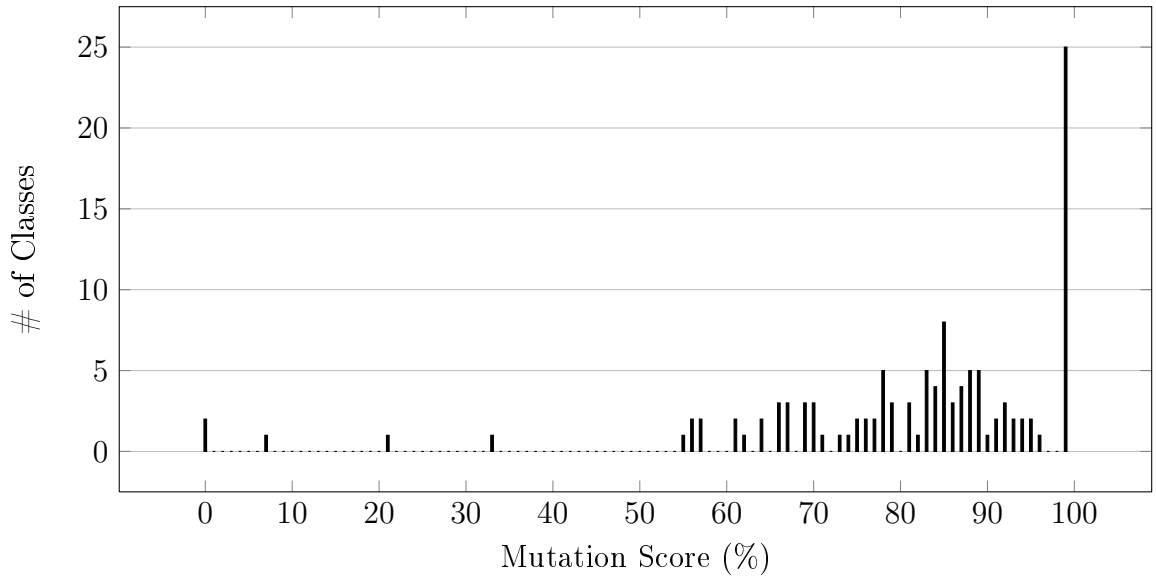


Figure A.15: Mutation score distribution of classes from *openfast* that can be used for training.

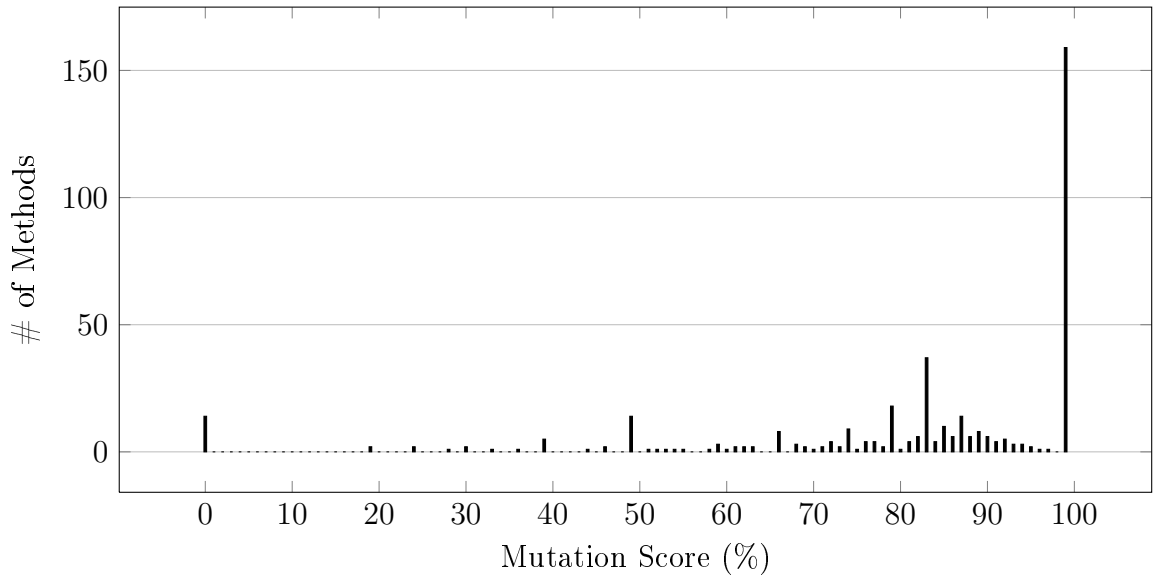


Figure A.16: Mutation score distribution of methods from *openfast* that can be used for training.

Appendix B

Feature Selection

In Section 5.2 we mentioned the we can reduce the used feature sets using a technique call *feature selection*. With feature selection it is possible to minimize the loss or possibly increase the predictive performance, and reduce cost (with respect to time and space). This appendix was originally planned for the optimization section (Section 4.3.4) though we decided to move this section in an appendix as the results were not significant.

In machine learning one typically gathers as many features as possible to supply sufficient data such that the learning algorithms can make accurate predictions. The general problem is to predict the correct category based on a vector, in some cases there are redundant, irrelevant or detrimental features to the predictive efforts. With the right set of features, the prediction performance can improve, or remain the same with less information required. With a reduced feature load, the actual performance (i.e., with respect to computational resources required) will improve. Feature selection makes it is possible to utilize a subset of the initially defined feature set that improve/maintain the predictive performance while requiring less data [GE03, BL97].

There are several approaches to feature selection, in particular *filter* and *wrapper*. Filters assess the quality/merits of features solely from the data alone as a preprocessing step [JKP94, BL97]. Various algorithms and measures can be used as a filter (i.e., information gain [GE03], correlation [Hal99], *FOCUS* [AD91], *Relief* [KR92], etc...). An alternative to filters are wrappers, which evaluates the actual performance of features using the classifier. Wrappers treat the classifier as a black-box and assess the performance using various subsets of features by using the actual predictor [JKP94, BL97]. Wrappers provide a more accurate and effective means in finding appropriate features, though inefficient as the classification process must occur many times using cross-validation with different features [KJ97].

We have over 3000 vectors (with undersampling in effect) with 15 features for the method-level data set (see Table 3.1). A wrapper approach could be very costly in our situation, though might prove more effective as found by Kohavi and John [KJ97]. For our research we decided against a wrapper approach due to the high computational cost involved. As an alternative to a wrapper approach we use Hall's *Correlation Based Feature Selection (CFS)* filter which is based on the follow definition: *"A good feature subset is one that contains features highly correlated with (predictive of) the class, yet uncorrelated with (not predictive of) each other"* [Hal99]. We used Hall's CFS implementation found in the machine learning toolkit *WEKA* [HFH+09].

To further investigate Hall's filter we created a correlation matrix of our features along with the raw mutation score and used category for the source code unit. We discovered that none of our features are highly correlated with the predicted category. In the class-level correlation there were only six features (i.e., *APAR*, *ATNBD*, *ATVG*, *NOF*, *NSC* and *SPAR*) that had a correlation between 0.3 and 0.5 (i.e., moderate correlation) with the rest being weak or no correlation. In the method-level correlations *ATMLOC* was the only feature with a moderate correlations while the rest were weak

or no correlation. There were a number of features that are highly correlated with each other (e.g., size and complexity). These findings suggest either:

- The selected features are insufficient in describing the predicted category.
- The difficulty of predicting the mutation score category is a highly complex process.

We believe that the observed correlations suggest the the prediction of the mutation score category is difficult. As mentioned in Section 3.1, there are two source artifacts involved in determining the mutation score, and our features are both well established descriptive metrics of these source artifacts.

We used the available data of the *all* data set (i.e., all the test subjects together) for both class- and method-level. Ten different undersampled data set of vectors that contain all our features were apply the CFS filter to produce an ascending order of features with respect to their correlation ranking (i.e., how effective the feature is with respect to others). To account for the undersampling we apply the CFS filter on each of the ten undersampled sets of data. We then use a simple rank summation to tally the results (i.e., ascending rank n has a value of n, and so forth), which then allowed us to create overall ranking of the features across the ten different undersampled data sets. We then removed the least useful feature one at a time and observed the new 10-fold cross-validation performance of the classifier using a subset of all the features.

The class-level cross-validation accuracy of the iteratively excluded features is shown in Figure B.1. We can see an interesting trend from the iterative exclusion of features, there is variation yet the mean accuracy remains approximately the same for about 17 iterations. After 17 iterations the cross-validation accuracy drops, which suggests that it is possible to exclude 17 features and still have approximately the same mean accuracy as with all the features. With respect to feature selection, an

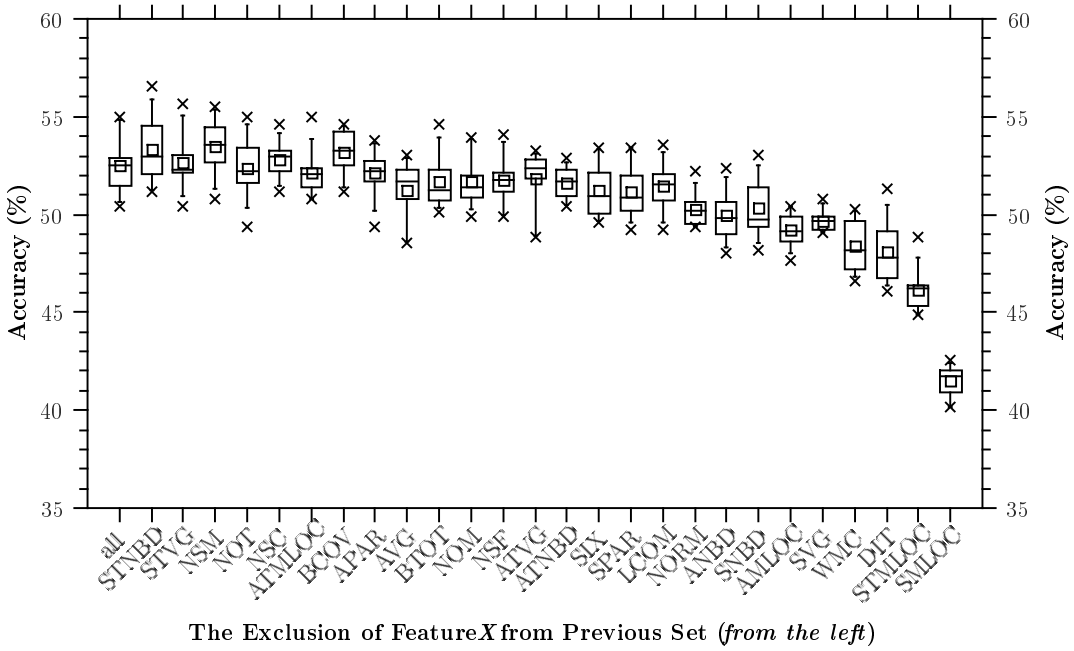


Figure B.1: Class-level cross-validation accuracy on the *all* subject over an iterative exclusion of features

The last feature not removed is 'NOF'.

---

ideal situation would allow use to use a reduced set of features that actually increase the cross-validation accuracy (i.e., by removing detrimental features). In our case we did not see any substantial increase in cross-validation accuracy, though we did not lower the mean accuracy over 17 iterations of exclusions. Another ideal situation is to completely remove certain feature sets, thus freeing us from the collection of these features. We can completely remove the coverage metrics (feature set ②) as all those metrics are excluded through the iterations. We need to keep in mind that CFS is a filter that removes features based on correlation with the category yet not with each other. An excluded feature might not necessarily be a *bad* feature, it might just be redundant. In the case of class-level features we can see that $STNBD$ and $STVG$ are the first two features excluded using CFS which makes sense considering both their correlation with the second last feature excluded ($STMLOC$) is above 0.995.

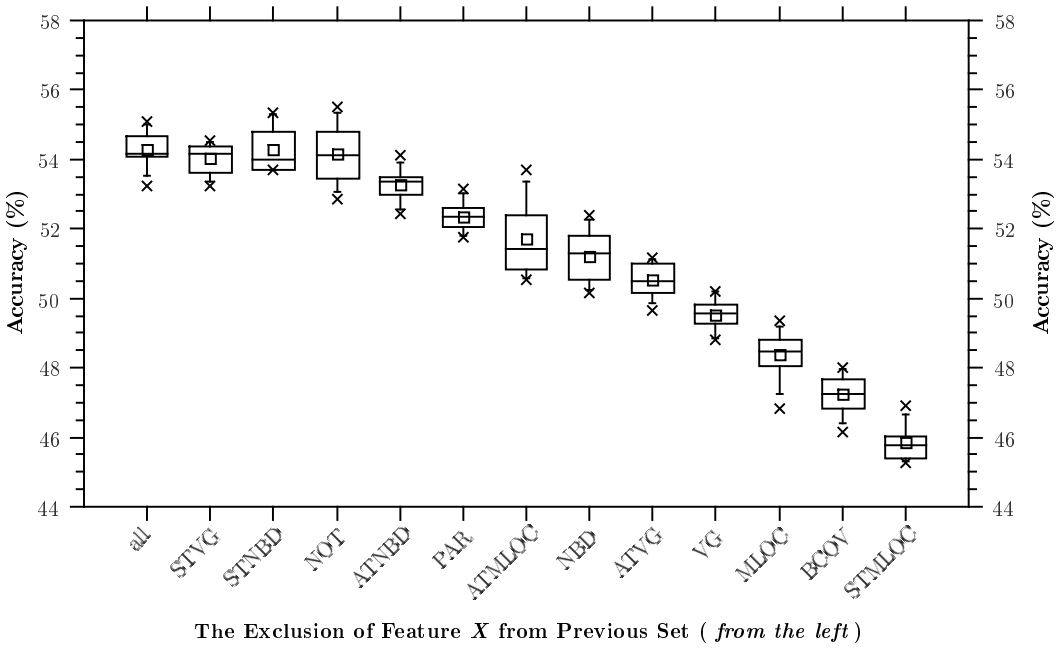


Figure B.2: Method-level cross-validation accuracy on the *all* subject over an iterative exclusion of features

The last feature not removed is 'BTOT'.

One interesting note here is that the last three features are each from three different feature sets (*STMLOC* belongs to ④, *SMLOC* belongs to ③ and *NOF* belongs to ①), which reinforces that each feature set is crucial to prediction and that the other features within these sets might be redundant.

The method-level cross-validation accuracy of iteratively excluding features (see Figure B.2) follows a similar trend to that of the class-level. If we consider the same approach as in the class-level we could potentially remove the first three features, which maintains the mean accuracy with a lesser amount of features. Unfortunately, it is not possible to completely remove a feature set with the exclusion of the first three features. Also we can see a similar trend in the order of excluded features that *STVG*, *STNBD* and *NOT* are removed early on with *STMLOC* and *MLOC* (i.e., similar to *SMLOC* from the class-level) being the last ones removed again due to high

correlation between these features. The last four features for method-level contain one feature from each feature set (i.e., the applicable ones for method-level, which excludes ③), which again reinforces the necessity of these feature sets.

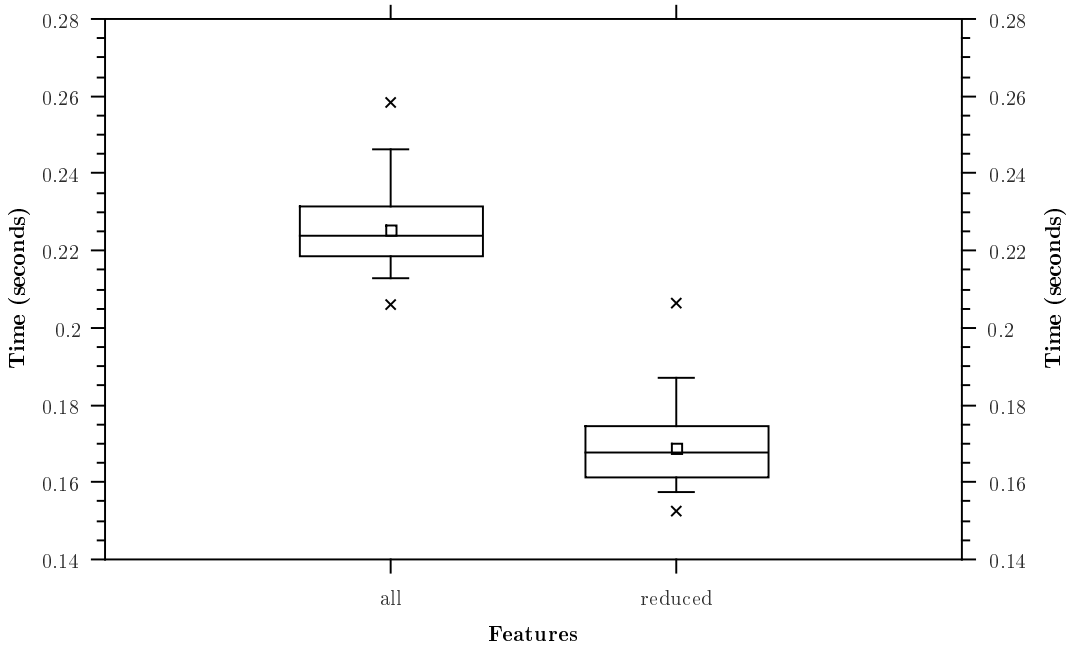


Figure B.3: The time required in seconds for class-level training and predicting using all features vs. a reduced set of features.
The reduced set of features correspond to the exclusion of the 17 left-most features from Figure B.1.

We were unable to demonstrate any substantial prediction performance gain in terms of cross-validation accuracy through feature selection, we decided to observe it from a resource perspective. Using 100 executions Figure B.3 and B.4 show the time required for training and predicting. We measured the time taken of training and predicting using both the reduced set of features as well as all the features to understand the performance gains with respect to time. We avoided measuring the time required with cross-validation as there are many more factors involved (i.e., the *easy script*, scales and grid searches using the data). With the training process we utilized the defaults that LIBSVM suggests for the *cost* (i.e., `1`) and *gamma* (i.e.,

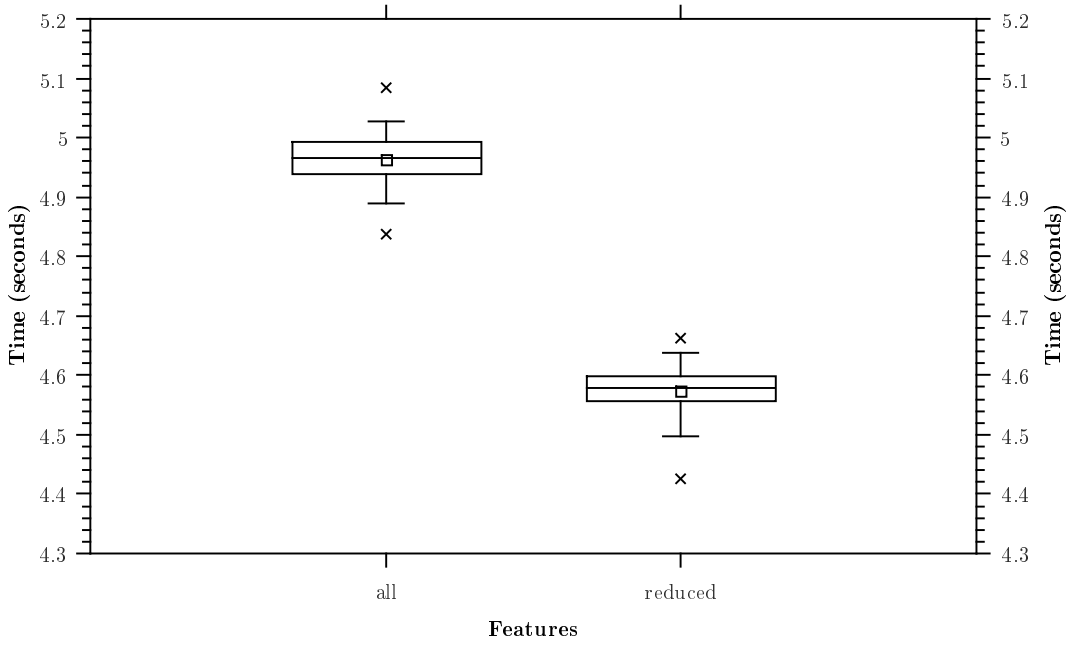


Figure B.4: The time required in seconds for method-level training and predicting using all features vs. a reduced set of features.

The reduced set of features correspond to the exclusion of the 3 left-most features from Figure B.2.

1/<number_of_features>) parameters. Regardless of the parameters chosen the relative ratio between the two sets of features will remain the same. We found that the reduced set of features reduces the training and prediction time for class-level by 25.1% and method-level by 7.9%. As the reduced set in class-level excluded more features than the method-level there is a great reduction in time taken for training and prediction.