Mutation testing: clustering mutants

Rasjaad Basarat
rasjaad.basarat@student.uva.nl

November 24, 2021 62 pages

Academic supervisor: Ana Oprescu, a.m.oprescu@uva.nl
Daily supervisor: Thomas Biesaart, thomas@infi.nl
Host organisation: Infi, https://infi.nl

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Abstract

Software testing is a crucial part of the software engineering process. A part of software testing is building test suites which contain unit tests. These unit tests are written by developers. As projects grow the test suite grows along. Maintaining and monitoring these test suites is important as they influence the cost of maintenance. For example, a project with a smaller test suite may have a higher maintenance cost as the effect of change has to be measured manually, which can be a time consuming process. Monitoring the quality of test code has shown that tests with lower quality lead to more defect-prone production code. A technique to measure the quality of test code is by measuring the test effectiveness.
Measuring the effectiveness of test suites detecting faults is part of the monitoring process. A technique to measure the test effectiveness is mutation testing. Mutation testing is a computationally expensive technique that requires the ability to run tests. Different techniques have been proposed to reduce the cost of mutation testing. One of these techniques is the clustering of mutants. By clustering mutants we can execute less mutants to reduce the cost.
Our research consists of two parts in which we cluster mutants. The first part consists of a white box approach and the second part of a black box approach. We conducted a state of the art comparison between three mutation testing tools for Java. Based on this research we selected a mutation testing tool to use in our research.
We identified characteristics to represent the mutants such that we can cluster them. For our white box approach we used hierarchical clustering and for our black box approach we trained the fuzzy c-means model. We calculated a weighted mutation score and compared this with the mutation score of a full set of mutants executed. Results show that with hierarchical clustering we can reduce the amount of mutants executed while maintaining the effectiveness. The clusters generated by the machine learning model were less accurate and showed a significant decrease in effectiveness. The machine learning model did not perform as well as the hierarchical clustering. However, more can be done by researching different approaches for training the FCM model and finding the right parameters.

Contents

Chapter 1

Introduction

Software testing is a critical part in the software engineering process. It is a standardized in the industry and is used for quality assurance[1]. Software testing can detect software bugs during the development process and can also serve for regression purposes[2]. During the software testing process developers write automated tests called unit tests. These tests can be run by testing frameworks such as JUnit and TestNG[3, 4]. Monitoring the quality of test code has shown that tests with lower quality leads to more defect-prone production code[5]. Research has also found that if the production code grows, the test code grows along with it[6]. Quality control on test suites is therefore important as the maintenance on tests can be difficult and generates risks if done incorrectly[7].
The growing complexity and size of code bases consequently lead to incomprehensible tests. As a result test bugs can occur. A test bug is a test that fail even tough the program is correct(*false positive*). Another bug, which is even worse, are tests that do not fail when the program is not working as desired(*false negative*). The *false negative* is a problem when breaking changes are not detected by the test suite.
To solve this issue we can measure the fault detecting capability of a test suite, this is called the test suite effectiveness. Test suite effectiveness is measured by the number of faulty versions of a System Under Test (SUT) that are detected by a test suite. Real faults are unknown in advance, we can use mutation testing as a proxy measurement. Existing research has found statistically significant evidence which correlates mutant detection with real fault detection[8].
Mutation testing tools create small faulty versions of the program. The test suite is then run to verify if these faults are detected. These faults are called mutants and are created by mutators. Mutators mutate specific expressions or statements in the source code. To prevent changing the overall functionality of the program, mutants are represented as very small changes[9].
After all the mutants are executed the mutation score can be calculated. The mutation score is the percentage of mutants killed divided by the total amount of mutants executed. If a large amount of mutants survives, it might be an indication that the quality of the test suite is poor as programming errors remain undetected. Mutation testing has considerable drawbacks[10], such as the equivalent mutant problem and being resource expensive.

1.1 Problem statement

Mutation testing requires a significant amount of computational resources[11]. The high cost requirement is often a barrier for adopting mutation testing [10]. A lot of techniques and methods have been developed to improve the performance, however, most of these approaches are not as effective as mutation testing a full set of mutants[10, 12].
A technique to reduce the number of mutants or the number of test case executions is mutation clustering. Mutation clustering aims to reduce the amount of mutants to be executed by clustering mutants[13, 14]. The clustering of mutants has been researched with promising results[13, 15, 16]. For example Ma et al., [13] clustered the mutants for expressions. However in their paper they indicate that the choosing expressions as the domain of their cluster is a limitation. A recurring problem in the research of mutation clustering is to determine the means of domains as the centre of a cluster[13, 15–17].
To reduce the cost of mutation testing, we try to find a solution that can cluster mutants while maintaining the accuracy of the same complete set of mutants. Our goal is to cluster any mutant that is generated within the Java programming language. This research aims to remove the scoping limitations present in existing research. There should be no requirements for which mutants can be clustered. We

do this with two different techniques: qualitative and quantitative. We devise a white and black box approach for clustering mutants. Our white box approach contains a qualitative analysis on mutants and a methodology to cluster them. The black box approach makes use of a machine learning model to cluster mutants. We structure our research on the following research questions:
Research Question 1: What set of characteristics can we identify for clustering generated mutants to reduce the amount executed, while maintaining effectiveness?
Research Question 1.1: How do the existing mutation testing tools for Java compare to each other?
Research Question 2: How can we train a machine learning model to recognize and cluster generated mutants to reduce the amount executed, while maintaining effectiveness?
We start with comparing existing mutation testing tools. We will use the existing literature as a starting point and research what would be the best tool to use for this thesis.
After we have selected a tool we will research existing clustering techniques. Based on the results of this research we select and implement a clustering algorithm that will cluster one ore more characteristics of the mutants. We will randomly select on mutant per of each cluster to be executed by the mutation tooling. We will then compare the mutation score of the clustered mutants with the mutation score of a full set of mutants.
To answer our second research question we will train a machine learning model to cluster mutants. We can then repeat the comparison we did for research question one.
We want to see how the algorithm/model performs on big and small projects. These project must also require a test suite.
We chose to focus on projects that use Java as main programming language because it is one of the most popular languages[18].

1.2 Contributions

Our research makes the following contributions:

1. A white box methodology to cluster generated mutants based on chosen characteristics to reduce the cost of mutation testing.
2. A black box methodology to cluster generated mutants based on chosen characteristics to reduce the cost of mutation testing.
3. A proof-of-concept (POC) which implements the qualitative methodology chosen and elaborated within the thesis.
4. A proof-of-concept (POC) which implements the quantitative approach to cluster mutants and is elaborated within the thesis.

1.3 Outline

In Chapter 2 we describe the background information of this thesis. Chapter 3 compares the different mutation testing tools to decide on a tool to use for this research. In chapter 4 we identify characteristics to represent mutants and design an experiment to cluster them. In chapter 5 we select a machine learning model and design an experiment to cluster mutants with this model. Results are shown in Chapters 7.1 and 7.2 for the experiments described in Chapters 4 and 5 respectively. The results are then discussed in Chapter 8.
Chapter 9, contains the work related to this thesis. Finally, we present our concluding remarks in Chapter 10 together with future work.

Chapter 2

Background

This chapter presents the necessary background information for this thesis. First, we define some basic terminology that will be used throughout this thesis. This thesis relies on clustering techniques. We review the state of the art on clustering algorithms and secondly the state of the art on machine learning models specialized in clustering.

2.1 Terminology

Mutation testing a form of white box testing in which testers change specific components of an application's source code to ensure a software test suite will be able to detect the changes. **Levenshtein distance** the minimum number of single-character edits (insertions, deletions or substitutions) required to change one piece of text into the other. **Maven** build tooling for Java software projects. **Gradle** build tooling for Java and Android software projects. **Mutator** definition of how the original source code is transformed into a mutant. **test engine** a test engine facilitates discovery and execution of tests. Examples for Java are TestNG and JUnit. **weighted mutation score** mutation score calculation designed to be comparable with the mutation score of a full set(see Chapter 6.3 for a detailed explanation).

2.2 Statistics

The terminology of statistics may lead to confusion if not understood correctly. Thus we discuss some basic terminology and principles in statistics. When performing a statistical test we have two hypotheses; a null hypothesis and an alternative hypothesis. A test contains a specific null and alternative hypothesis. The goal of such a test is to reject the null hypothesis. Rejecting a null hypothesis can be done by looking at the resulting *p-value*. The *p-value* is the chance that the value of the statistical test occurs if the null hypothesis is true on a zero to one scale. We can reject a null hypothesis if we think that this chance is below a certain threshold. This threshold is determined beforehand and named the *alpha-value*. A commonly used value for alpha is 5%. We cannot reject the null hypothesis if the *p-value* is below the *alpha-value*. This does not necessarily mean that the null hypothesis is true. This can have different reasons, for example, the data set used to validate the hypothesis is too small.

2.3 Clustering algorithms

This research focuses on clustering mutants and use of existing clustering algorithms. Many different types of clustering algorithms have been proposed[19]. While there is a lot of diversity, some methods are more frequently used than others[20]. Rodriguez et al., compared the performance of nine different clustering algorithms. We provide a summary of these algorithms in the following paragraphs.

K-means

The K-means algorithm identifies k number of centroids, and then allocates every data point to the nearest cluster, while keeping the centroids as small as possible[21].

CLARA

CLARA uses multiple fixed samples of the data set to minimize sampling bias and, select the best medoids among those samples[22]. A medoid is defined as the object i for which the average dissimilarity to all other objects in its cluster is minimal[22].

Hierarchical clustering

Hierarchical clustering groups similar objects into groups called clusters[23]. The endpoint is a set of clusters or groups, where each cluster is distinct from each other cluster, and the objects within each cluster are broadly similar to each other[23].

EM

EM(expectation maximization) clustering technique is similar to the K-Means technique[24]. Instead of assigning examples to clusters to maximize the differences in means for continuous variables, the EM clustering algorithm computes probabilities of cluster memberships based on one or more probability distributions[24].

Spectral clustering

Spectral clustering is a technique with roots in graph theory. It identifies communities of nodes in a graph based on the edges connecting them[25]. Spectral clustering uses information from the eigenvalues of special matrices built from the graph or the data set to perform dimensionality reduction before clustering in fewer dimensions[25]. The similarity matrix is provided as an input and consists of a quantitative assessment of the relative similarity of each pair of points in the data set[25].

Subspace

Subspace clustering algorithms consider the similarity between objects with respect to distinct subsets of the attributes[21]. Different subsets of the attributes might define distinctions between each other. The algorithm can identify clusters that exist in multiple, possibly overlapping, sub spaces[26].

Dbscan

Dbscan is a density-based clustering non-parametric algorithm[27]. Given a set of points in some space, it groups together points that are closely packed together, marking outliers as points that lie alone in low-density regions[27].

OPTICS

The OPTICS(Ordering Points To Identify the Clustering Structure) algorithm starts with a data point and expands its neighborhood using a similar procedure as in the Dbscan algorithm[28]. The difference is that the neighborhood is first expanded to points with low core-distance[28]. The optics algorithm can detect clusters having large density variations and irregular shapes[28].

The goal of their study was to guide researchers, who have little experience in data mining techniques, to the application of clustering methods. They evaluated the algorithms in three distinct situations: default parameters, single parameter variation and random variation of parameters. They used 400 generated artificial data sets which were normally distributed.
The results reported in their research are respective to specific configurations of normally distributed data and algorithmic implementations. Nonetheless they do give a good overview on how the algorithms compare to each other.
For the default parameter experiment, the spectral clustering algorithm had the best performance and the hierarchical algorithm had the worst performance.
Regarding single parameter variations, for data sets containing 2 columns, the hierarchical, optics and EM methods showed significant performance variation.
With respect to the multidimensional analysis for data sets, the performance of the algorithms for the multidimensional selection of parameters was similar to that using the default parameters. They conclude their research with observing that, for data sets with 10 or more columns the spectral algorithm

consistently provided the best performance. However the EM, hierarchical, k- means and subspace algorithms can also achieve similar performance with some parameter tuning. The optics and dbscan algorithms aim at different data distributions than performed in this study. There different results could be obtained for non-normally distributed data.

2.4 Machine learning models for clustering

There are many different types of machine learning models with their respective applications. For this thesis we will focus on machine learning models designed for clustering.
Machine learning algorithms are divided in supervised and unsupervised learning algorithms[29]. Supervised learning is the machine learning task of learning a function that maps an input to an output based on example input-output pairs[29]. In contrast to supervised learning, unsupervised learning shows self-organization that captures patterns as neuronal predilections or probability densities[29]. Unsupervised learning algorithms are commonly used for classification and categorization[29]. Unsupervised learning is a type of algorithm that learns patterns from untagged data[29]. These machine learning algorithms are commonly used for clustering and dimensionality reduction[29].
K.L. Du surveyed different machine learning models used for clustering[30]. We provide a summary of these models in the following paragraphs.

Self organizing map.

The self organizing map (SOM) is a neural network-based dimensionality reduction algorithm generally used to represent a high-dimensional data set as two-dimensional discretized pattern[30]. Reduction in dimensionality is performed while retaining the topology of data present in the original feature space[30]. The self organizing map computes the Euclidean distance of the input pattern x to each neuron k, and finds the winning neuron using the nearest-neighbor rule[30] . The winning node is called the excitation center.

Fuzzy C-means clustering.

Fuzzy C-means clustering works by assigning membership to each data point corresponding to each cluster center on the basis of distance between the cluster center and the data point[30]. The more the data is near to the cluster center more its membership towards the particular cluster center. The weighting parameter m is called the fuzzifier[30]. M determines the fuzziness of the partition produced, and reduces the influence of small membership values[30].

Learning vector quantization.

One or more prototypes are used to represent each class in the data set, each prototype is described as a point in the feature space[30]. New (unknown) data points are then assigned the class of the prototype that is nearest to them[30]. In order for nearest to make sense, a distance measure has to be defined[30]. An example of such a distance measure is the Euclidean distance. Learning vector quantization(LVQ) is a supervised classification model. The LVQ can be treated as a supervised version of the SOM[30].

Mountain and subtractive clustering.

The mountain and subtractive clustering models are an effective method for estimating the number of clusters[30]. The subtractive clustering model is an extension of the mountain model[30].
The method grids the data space and computes a potential value for each grid point based on its distance to the actual data points. Each grid point is a potential cluster center. The potential for each grid is calculated based on the density of the surrounding data points. The grid with the highest potential is selected as the first cluster center and then the potential values of all the other grids are reduced according to their distances to the first cluster center[30].

Neural gas.

The neural gas is a topology-preserving network, and can be treated as an extension to the C-means[30]. A data optimal topological ordering is achieved by using neighborhood ranking within the input space at each training step[30]. To find its neighborhood rank, each neuron compares its distance to the input

vector with those of all the other neurons to the input vector[30]. Unlike the self organizing map(see Chapter 5.1.6, which uses predefined static neighborhood relations, the neural gas network determines a dynamical neighborhood relation as learning proceeds[30]. A benefit of the neural gas network is that it is able to spawn neurons. This means that it is not necessary to know the number of clusters in advance.

ART networks.

ART models are characterized by systems of differential equations that formulate stable self-organizing learning methods[30]. The stability and plasticity properties as well as the ability to efficiently process dynamic data make the ART attractive for clustering large, rapidly changing sequences of input patterns[30].

K.L. Du describes the mathematical principles on which the machine learning algorithms are based as well as the origin, pros and cons of each model.
He also touches on subjects relevant to clustering and machine learning models such as the under-utilization problem, fuzzy clustering, robust clustering, clustering based on non- Euclidean distance measures, supervised clustering, and hierarchical clustering. Machine learning model variants and their references are also discussed.
He closes his paper with computer simulations of some of the machine learning models he surveyed.

Chapter 3

Mutation testing tools

To cluster mutants we need to select a tool that generate mutants and executes mutants. Such tools are called mutation testing tools. Our goal is to find the tool that generates the most amount of mutants. First, we review existing literature. Second, we extend the comparison by adding information about the current state of the tools. With the combination of the information gathered we make a decision on what tool to use for this thesis.

3.1 State of the art documented

The literature used several criteria for comparing mutation testing tools for Java:

- Execution time of each tool.
- The cost as the number of test cases that needs to be generated and the the number of equivalent mutants that would have to be inspected.
- Effectiveness of the mutation adequate test suite of each tool. Every test of the suite adds to the effectiveness score. If a test is removed less than 100% effectiveness can be achieved. To evaluate the effectiveness of each tool's mutation adequate test suite, a cross-testing technique is applied. The adequate test suite per tool is run on the set of mutants generated by the other tools.

Kintis et al. analysed and compared the performance of PIT, MuJava and Major[31]. They started out with performing a mini literature survey on mutation testing tools used in existing research on test effectiveness in Java. With a data set of twelve methods and six Java projects they evaluated each tool using a cross-testing technique. As a result they found that the most effective mutation adequate test suite was from MuJava, followed by Major and PIT. The score of the application cost was the inverse of the previous ranking: PIT generated the smallest set of equivalent mutants and required the least test cases. During their research they also compared the different mutant operators the tools supported. An overview of mutant operators(also known as mutators) per tool was created. They compared the mutant operators and reported overlapping operators.
Marki and Lindstrom performed their research on the same three mutation testing tools.[32]. The same cross-testing technique used by Kintis et al., was applied for three small Java programs. These programs are popular programs used in testing literature. They found that the mutation tools do not subsume each other. "one mutant subsumes a second mutant if every test that kills the first mutant is guaranteed also to kill the second [33]" According to the research a ranking of the tools would be as follows; MuJava generated the strongest mutants followed by Major and PIT. Furthermore they found that MuJava generated significantly more equivalent mutants[32]. MuJava also had an execution time twice the amount of that of Major and PIT combined.
Laurent et al., introduced PIT+, a version of PIT with extended set of mutators[34]. They used the same tests suites that were generated by kintis et al., and combined these into an adequate test suite that would detect the combined set of mutants generated by PIT, MuJava and Major. They discovered that the set of mutants generated by PIT+ was stronger than the combined sets generated by the other three tools. Fortunately the extended set of mutants used in PIT+ is now integrated in the PIT project[35].

3.2 State of the art tools

The literature is clear about how the mutation testing tools compare to each other. However the most recent study is from 2017. At the time of writing this thesis these studies are at least four years old. Some of the tools were updated or are still under active development[36, 37].
There are three candidate tools; Major, MuJava and PIT. We extend the existing mutant operator comparison for these tools with the operators that have been added since the publishing date of the literature. We also research the overlap for the new mutant operators. Table 3.1 shows the mutant operators that were developed after the publication of the literature. The amount of mutant operators for MuJava and PIT have increased significantly while there were no new mutant operators for Major.

MuJava		**PIT**	
IHD	Hiding variable deletion	EM	Empty Returns
IHI	Hiding variable insertion	FR	False Returns
IOD	Overriding method deletion	NR	Null Returns
IOP	Overriding method calling position change	TR	True Returns
IOR	Overriding method rename	PR	Primitive Returns
ISI	Super keyword insertion	ER	Experimental Switch
ISD	Super keyword deletion	BI	Big Integer
IPC	Explicit call to a parent's constructor deletion	NRC	Naked Receiver
PNC	New method call with child class type	N	Negation
PMD	Member variable declaration with parent class type	AOR	Arithmic Operator Replacement
PPD	Parameter variable declaration with child class type	AOD	Arithmic Operator Deletion
PCI	Type cast operator insertion	CR	Constant replacement
PCC	Cast type change	BO	Bitwise Operator
PCD	Type cast operator deletion	ROR	Relational Operator Replacement
PRV	Reference assignment with other comparable variable	UOI	Unary Operation Insertion
OMR	Overloading method contents replace		
OMD	Overloading method deletion		
OAC	Arguments of overloading method call change		
JTI	This keyword insertion		
JTD	This keyword deletion		
JSI	Static modifier insertion		
JSD	Static modifier deletion		
JID	Member variable initialization deletion		
JDC	Java-supported default constructor deletion		
EOA	Reference assignment and content assignment replacement		
EOC	Reference comparison and content comparison replacement		
EAM	Acessor method change		
EMM	Modifier method change		

Table 3.1: Current mutators supported.

Tools sometimes implement the same mutator differently, resulting in different mutant sets. Table 3.2 shows an overview over the relations between the mutators of MuJava and PIT. Some of the new mutators from the tools map to the already existing mutators of the other tools. To get a complete overview we include the mutators that have been reviewed in the studies of Kintis et al., and Marki and Lindstrom.

MuJava mutator		**PIT mutator**	
AORB	Arithmetic Operator Replacement Binary	M	Math
ASRS	Short-Cut Assignment Operator Replacement		
SOR	Shift Operator Replacement		
COR	Conditional Operator Replacement		
AOIU	Arithmetic Operator Insertion Unary	UOI	Unary Operation Insertion
AOIS	Arithmetic Operator Insertion Short-cut		
ODL	Operator Deletion	AOD	Arithmic Operator Deletion
AODS	Arithmetic Operator Deletion Short-cut	RI	Remove Increments
ROR	Relational Operator Replacement	CB	Conditionals Boundary
AORS	Arithmetic Operator Replacement Short-Cut	I	Increments
COD	Conditional Operator Deletion	RC	Remove Conditionials

Table 3.2: Overview of mutator overlaps between MuJavav and PIT

Next we try to generate mutants with all three tools. All of the sources that we used for generating mutants are build with the build tools Maven or Gradle. Marki and Lindstrom[32] mention that Major includes support for Maven, but it is not documented anywhere. For this reason we can not generate mutants with Major. Table 3.3 shows the amount of mutants generated per project and per tool. The results show that in all of the cases PIT generates more mutants than MuJava. In some cases the difference is significant. This difference can partially be explained by the fact that MuJava does not support source projects with Java version 1.7 or higher[38]. All the code that uses the of features from this version of Java and above resulted in an error during mutant generation.

	version	**LOC (Lines of code)**	**MuJava**	**PIT**
Commons-numbers.core	1.0-beta1	450	988	4384
Jodatime	2.10.10	28811	52925	112772
Zxing 3.4.1	3.4.1	24792	65983	161409
Google Auto Common	0.11.0	2338	20	5219
Google Auto Factory	1.0-beta9	1507	69	5832
Google Auto Value	1.7.5	7466	745	16746
Google Auto Service	1.0-rc6	750	0	475
ScribeJava-Core	8.1.0	5709	1358	5746
Checkstyle	8.41.1	38491	4529	100952
Fastjson	1.2.75	43405	70577	116188
Jfreechart	1.5.3	91876	0	350741
Commons-lang	3.12.0	29836	31145	134764
Commons-codec	1.15.0	9656	21719	54804
Commons-text	1.9.0	9781	10403	48490
Commons-io	2.8.0	13947	13245	44631
Gson	2.8.6	8078	9198	28485
Commons-cli	1.4.0	2782	831	7193
Commons-csv	1.8.0	1855	2081	6906

Table 3.3: Amount of mutants generated by MuJava and PIT

3.3 Selecting mutation testing tool

We studied literature on three different tools and extended the research with the current state of the art. MuJava is not actively maintained and has not been updated in the last few years. It does not support JUnit 4 and all versions of TestNG[38]. These are test engines and are crucial for executing tests. MuJava also does not support source projects with java version 1.7 or higher[38]. Conforming to these requirements would reduce the set of projects we could use in or experiments.
While Major supports JUnit 4, we did not succeed in generating mutants with this tool. It would be too time consuming to customize all source projects to work with Major. Furthermore there is not much documentation available for this tool.
PIT targets the industry, is actively maintained and is open source[31]. It supports Maven, Gradle, has a Command Line Interface (CLI) and has a faster execution time than the other tools. For example PIT provides a plugin system in which you can inject your own code in various stages of mutation testing process[39]. PIT also generated significantly more mutants in every project we have tested. Based on the information presented in this chapter we decide that PIT is the best choice for generating and executing mutants.

Chapter 4

Qualitative clustering approach

The following sections describe the white box approach for clustering mutants. We perform a qualitative analysis on mutants to identify characteristics. These characteristics represent the mutants. With the mutants represented we devise a methodology for clustering mutants. Our goal is to cluster every mutant that is generated while maintaining effectiveness.

4.1 Identifying characteristics

Our goal is to cluster any mutant that fit the requirements of the selected mutation testing tools. We want to solve the scoping problem in existing research[13, 15–17]. To achieve this we need to identify characteristics that every mutant contains. As part of our white box methodology we perform a qualitative analysis on mutants and identify characteristics that are relevant for defining the means of domains as the centre of clusters.

4.1.1 Overview of characteristics

Zhang et al., identifies several mutant characteristics[40]. The goal of their research is detecting as accurate as possible whether a mutant survives or not before executing the mutant. The mutant characteristics identified by Zhang et al., have been used in other research with promising results[41]. While the goal of their research is different than ours, we can still use the characteristics they identified. We select characteristics relevant to our research from the research of Zhang et al., and Oonk[41]. We extend the list of characteristics with data we can extract from the selected mutation testing tool.
PIT generates and executes mutants on byte code level[42]. By mutating in Java byte code we can identify characteristics specific to Java byte code. PIT also gathers mutant metadata. We can extract this mutant metadata and identify characteristics. With the combination of information extracted from PIT and characteristics selected from existing research we identify the following characteristics:

1. Mutant similarity.
2. Amount of tests the mutant is challenged by.
3. Mutator identifier.
4. Mutant opcode.
5. Mutant return type.
6. Mutant contains local variables.
7. Mutant is in try catch block.
8. Mutant is in finally block.
9. Name of class that contains the mutant.
10. Name of function that contains the mutant.
11. Line number of the start of the block that contains the mutant.
12. Line number on which the mutation occur.

In the following sections we elaborate on these characteristics.

4.1.2 Mutant similarity

A piece of code is a textual representation of instructions for a computer. A mutant is a transformation of a piece of code. This transformation is predefined. The difference between a mutant and its parent can be defined as similarity. We identify the similarity between a mutant and parent as a characteristic. To measure this characteristic we select a similarity measure. There are existing similarities measures that base their similarity in different ways[43].
The Levenshtein distance defines the distance between two strings by counting the minimum number of operations(create/edit/delete) needed to transform one string into the other[43]. The more the textual representation of a mutant differs from its parent the bigger the Levenshtein distance will be. A unit test tests a small piece of code. The more this piece of code differs from its original the more likely it is the test will fail. The Levenshtein distance measures this difference. We select the Levenshtein distance as similarity measure for our research[43]. This could be a useful characteristic because mutants that have the same similarity may be grouped together.

4.1.3 Amount of tests challenged by mutant

A mutant may be challenged by tests. If the test fails, the mutant is considered killed. If the test passes, the mutant is considered survived. If the mutant is not challenged by any tests it is also considered survived. The amount of tests a mutant is challenged by is dependant on the coverage and test effectiveness of the test suite. We identify this number as a characteristic. Every test the mutant is challenged by is an occasion for the mutant to be killed or not. This could be a useful characteristic because mutants that have more chances to be killed may be grouped together.

4.1.4 Mutator identifier

At the time of writing the selected mutation testing tool PIT supports more than 100 mutators[44]. A mutator is the definition of how the code is mutated from its original[45]. Each mutator mutates the source code in a dfferent way. PIT assigns identifiers to their mutators for execution and reporting purposes. Each mutant has a mutator identifier that matches with how the mutant is mutated from the source code. We can extract this data from PIT and use it as a characteristic. It could be a useful characteristic because mutants that are changed in the same predefined way may be grouped together.

4.1.5 Mutant opcode

"A Java Virtual Machine instruction consists of a one-byte opcode specifying the operation to be performed, followed by zero or more operands supplying arguments or data that are used by the operation." (Java 12 SE docs ch.12). When PIT mutates the source code it uses Java opcodes to alter the original byte code. A record of which opcode is being used is stored in memory during the mutation testing process. This opcode can differ per mutator and is dependant on how the code is written. During the mutation testing process of PIT we can extract the opcode. We identify the opcode as a characteristic. The opcode is a characteristic that influences the end result of the mutant in byte code representation which in turn can influence other characteristics, for example the Levenshtein distance. The opcode could be a useful characteristic because mutants with the same opcode may be grouped together.

4.1.6 Mutant return type

A mutant may contain a return statement. This is dependant on the source code that is mutated. A return statement can be identified as a characteristic. If a mutant contains a return statement, we can extract its type from PIT. There is the possibility that a mutant does not contain a return statement, in this case we set the mutant return type to a default value. The return type is a distinctive property of a piece of code, this distinction a useful characteristic because mutants with the same return type may be grouped together.

4.1.7 Mutant location

The location of a mutant in PIT is defined by multiple properties; the name of the class, the name of the function, the line number of the start of the block the mutant is located in and the line number the mutation occurs on. These properties refer to the location in Java byte code. The location can be

identified as a characteristic of a mutant. PIT uses this location during the mutation testing process and for reporting purposes. We can extract the location as defined by the properties in PIT. Each property is identified as a characteristic. The location of a piece of code is a distinctive property. The same piece of code in a different location may have a different effect on the source code or unit tests. For example two mutants with the same byte code may have a different result in two different locations of a software project. This is dependent on different factors one of which being the unit tests the mutant is challenged by. A mutant may not be unique if the location is not taken into account. It could be a useful characteristic because mutants that are in the same location may be grouped together.

4.1.8 Error handling capability

A technique to handle errors in Java is to use try-catch blocks[46]. The presence of try-catch blocks are an indication of the error handling capabilities of certain pieces of code. An addition to the try-catch blocks are finally blocks. A finally block is executed after a try and catch block[46]. There is a possibility that a mutant may be generated inside these blocks. If a mutant is located in a try, catch or finally block can be defined as a characteristic. It could be a useful characteristic because the mutants that are located in the same type of block may be grouped together.

4.1.9 Local variables

A mutant may be located in a location(see Chapter 4.1.7) that may contain local variables. There is a chance a mutant modifies a local variable. A variable modified by a mutant may increase the chance for a mutant to be killed. This is because a unit test may check for a specific value for a specific local variable. The amount of local variables a location contains can be identified as a characteristic. We can extract the amount of local variables a location contains from PIT. This could be a useful characteristic as mutants with the same amount of local variables may be grouped together. It could also be possible that there is a relation between the amount of local variables and the result of a mutant(killed or survived).

4.1.10 Extracting characteristics

To extract all the characteristics we identified we need to run PIT and configure it to generate all mutants that are possible within the tool. PIT works in phases, the first phase is the generation phase where mutants are generated. The second phase is the execution phase where mutants are challenged by the test suite. The last phase is the reporting phase where a report is generated based on the users preferences and data generated by PIT. During the various stages we can extract characteristics by using PIT's plugin system. The plugins are executed after each other and per class. In other words first plugin A will make changes or analyse all classes then plugin B etc.
PIT offers a plugin system in which developers can inject their own code in PIT[47]. There are two main types of plugins; a Mutation Result Listener and a Mutation interceptor[47]. A mutation result listener receives the details of analysed mutations as they arrive[47]. A Mutation interceptor is passed a complete list of all mutation that will be generated to each class before the mutation are challenged by tests[47]. The implementation of our experiment makes use of this plugins system.
We developed two plugins to extract characteristics. The first plugin is of type Mutation interceptor. It is provided a list of details per class. We can extract all characteristics but the Levensthein distance and the amount of tests a mutant is challenged by. To calculate the Levensthein distance we need the byte code of the original code and the mutated code. We generate the byte code of the original code based on the location provided in the list of details. We make use of existing functionality in PIT for generating byte code. The mutator is also provided in the list of details. With the combination of location and mutator we can generate the byte code of the mutant. With both the mutated byte code and original byte code available we can calculate the Levenshtein distance.
The characteristics gathered until this phase of the mutation testing process are written, together with a unique identifier, to a CSV file.
The second plugin is a mutation result listener. This plugin gets passed a list of the results per mutant. This list contains data about the mutant survival status. It also contains data about the number of tests the mutant is challenged by. We store this data together with the identifier to a different CSV file.
When the mutation testing process has finished we need to merge the characteristic gathered in from the plugins. We add the characteristic; the number of tests the mutant is challenged by to the data file containing all the other characteristics. All the files contain the same unique identifier per mutant on

which we can merge the data. We do not add the information about the survival of a mutant. We only need this data to validate our results.

4.2 Clustering mutants

We identified characteristics to represent mutants in Chapter 4.1. With this representation of the mutants we can devise a methodology to cluster them. The following sections describes the methodology for our white box approach of clustering mutants.

4.2.1 Hierarchical clustering

Clustering Levenshtein distances has been done before with hierarchical clustering with promising results.[48, 49]. Research shows that hierarchical clustering performs better when clustering with at least ten features[19]. It also states that varying the parameters of hierarchical clustering improves the performance compared to that of the default settings of the algorithm[19]. We identified more than ten characteristics and can use them as features. We adjust the parameters of the algorithm based on the characteristics in our data set. Bases on these observations and the research of Rodriguez et al., we select hierarchical clustering as it gives us the best performance in terms of partitioning. Next we explain the configuration we use for clustering the mutants.
Hierarchical clustering is subdivided into agglomerative and divisive. The agglomerative hierarchical technique follows bottom up approach whereas divisive follows top-down approaches. Hierarchical clustering uses different metrics which calculates the euclidean distance between two clusters and the linkage criteria[48]. The linkage criteria specifies the dissimilarity in the sets as a function of the pair-wise distances of observations in those sets[48]. We reviewed the different linkage criteria and concluded the following.
Research shows that the complete linkage outperforms the single linkage method[50]. The ward linkage and complete linkage methods perform the same when clusters are well separated[50]. However if the clusters overlap the ward linkage outperforms the complete linkage[50].
We identify all characteristics per mutant as a separate cluster. Starting out with each mutant as a separate cluster we can use the agglomerative form of hierarchical clustering. Since we cannot assume that our clusters are well separated we chose to use the ward linkage method for our clustering algorithm. In summary, we select the agglomerative hierarchical clustering algorithm with ward linkage to cluster mutants represented by the characteristics we gathered.

4.2.2 Levenshtein distance on Java byte code

A mutant is a piece of code that differs in a predefined way from its parent. Java source code compiles into Java byte code. During the compilation of Java code certain context is abstracted away[51]. Optimisations are applied which also changes the Java byte code[51]. As a result the textual similarity between a Java code mutant and its parent is different from the similarity between a Java byte code mutant and its parent. While the textual similarity may be different between the Java code and Java byte code the functionality remains the same. Java byte code reflects more of the semantic nature of the source code than the source code itself does. PIT generates and executes mutants on byte code level[42]. In other words PIT executes the unit tests of a source against the byte code of a mutant.
By calculating the distance on byte code we filter out the context that is present in Java code. This gives us a Levenshtein distance that represents more of the semantic difference, between a mutant and parent, than calculating the distance for Java code. For this characteristic we use the Levenshtein distance between a Java byte code mutant and its parent.

4.2.3 Levenshtein distance implementation

There are different implementations of the Levensthein distance. For our experiment we need to calculate the Levenshtein distance on the byte code generated in PIT. Since we make use of the plugin system we also choose to use an implementation that is written in Java. The Levenshtein distance can be calculated in different ways for example with recursion. The implementation with recursion is has a complexity of O(N^2) and a memory need of O(N^2). There is also an improved implementation that does not use recursion. While the complexity is the same the memory need is reduced to O(N). This last implementation is the implementation we selected for our experiment. The implementation described

above is also offered by a library. This library is the Apache Commons Text library[52], in our experiment we make use of this implementation for calculating the Levenshtein distance.

4.2.4 Categorical data

There are different categorical variable encoding techniques available[53]. The categorical characteristics we use have no particular ranking compared to each other. There is also no specific order to the characteristics. I.e. a return type void is not better or worse than a return type string. The same goes for the location characteristics, there is no location that should have a bigger weight than the other locations. The individual characteristics do contain a finite set of values. For example multiple mutants may contain the same class name. The hierarchical clustering algorithm needs all characteristics in a numerical form[50]. The characteristics mutator identifier, class name, method name and return type are non numerical. To deal with this problem we apply categorical variable encoding to these specific features. Taking into account the properties of our categorical characteristics the nominal variable encoding fits our requirements. Nominal encoding comprises a finite set of discrete values with no relationship between values[53]. Therefore we implement this type of encoding in the experiment.

4.2.5 Number of clusters

Agglomerative hierarchical clustering can continue to cluster until there is one cluster left. Naturally this cluster will contain all the mutants. We can cut off the clustering algorithm at any point. We decide on the number of clusters based on the amount of mutants generated in a full set. We select the number of clusters from a performance increasing perspective. Mutation clustering increases the performance by reducing the amount of mutants executed[10]. By reducing the amount of mutants, for example by half, we increase the performance by 50%[8]. As a starting point we perform three experiments where the cluster size is 25%, 50% and 75% of the total amount of mutants respectively. We can evaluate the results and can decide to cut off the clustering at different points. The amount of mutants inside a cluster is be decided by the clustering algorithm.

Chapter 5

Quantitative clustering approach

In the following sections we devise a black box approach for clustering mutants. We make use of the characteristics identified in Chapter 4.1. While our goal remains the same as in Chapter 4, the method to achieve it is different.
We start by selecting a machine learning algorithm. Next we select a training strategy and an implementation. Last we select the parameters required by the algorithm.

5.1 Selecting a machine learning model

In this section we select a machine learning model. We start by choosing between supervised or unsupervised learning. Then we look at the machine learning models that are discussed in the survey done by K.L. Du. Based on the choice of supervised or unsupervised learning we review the corresponding models and decide if they fit our use case by a process of elimination.

5.1.1 Supervised or unsupervised learning

In this subsection we decide on supervised or unsupervised learning. The application and description of these two types of learning are described in Chapter 2.
Supervised machine learning models require a labeled data set. We can argue that we can label our data set with the results of our first experiments. Each mutant would then be labeled with the identity of the cluster it is clustered in by the hierarchical clustering model. If we would train a supervised model with the data labeled as described, it may result in a model that can do the same as hierarchical clustering. Our research is not to prove if we can train a model to cluster mutants the same way as hierarchical clustering. We want to train a model to find the relations between the characteristics without having to define these relations manually.
Unsupervised learning models do not need a labeled data set and can recognize relations or discover hidden patterns in the characteristics we identified. This corresponds to what we want to achieve with this experiment. Therefore we choose unsupervised learning to continue with in our experiment.

5.1.2 Mountain and subtractive clustering

As described in Chapter 2.4 the mountain and subtractive clustering models calculate the amount of clusters needed for a specific data set. Subtractive clustering is designed for learning fuzzy systems models from data[54]. If it is required to find out the number of clusters we need for our model we can make use of these models.

5.1.3 ART models

While the ART models are good at what they do, they do not fit our purpose of clustering mutants. As described in Chapter 2.4 the ART models expect rapidly changing sequences of input patterns. Our data does not contain binary input patterns and it is also not rapidly changing. Therefore we do not select an ART model.

5.1.4 Neural gas

The neural gas network is a learning network. One of the properties of a learning network is that it is not designed to fit new data points[29]. For this reason we cannot use the neural gas network to predict in what cluster a new mutant should be placed. Therefore we do not select the neural gas network.

5.1.5 Learning vector quantization

The learning vector quantization model is a supervised learning model. We established that we will use an unsupervised learning model to cluster mutants. Therefore we do not select the supervised learning vector quantization model.

5.1.6 Self organizing map, C-means and fuzzy c-means

The self organizing map (SOM), C-means and fuzzy c-means (FCM) models are all models that would fit our purpose. To choose between these models we look at the performance of each model. Mingoti et al., compared the performance of these three models [55]. They did a Monte Carlo simulation where the cluster sizes and amount of random numbers in the data set varied each simulation[55]. Other variables such as cluster boundaries and overlap were also controlled variables in the experiment. They observe that the C-means and fuzzy c-means had good performance for non overlapping situations[55]. The best results for average recovery and internal dispersion rates were found for fuzzy c-means which was stable in all situations achieving recovery averages over 90%[55] The C-means method was affected by the presence of a large amount of outliers. They conclude that the SOM did not perform well in many cases being affected by the amount of variables and clusters even for the non overlapping cases[55]. They also observe that their results partially overlap with other research in the same domain.
Based the conclusions made by Mingoti et al., we can conclude that the FCM model has the best performance overall. Thus for this experiment we select the FCM model.

5.2 Training strategy

Our data set consists of mutants represented by their characteristics. The characteristics of the mutants are generated per project and have a project specific context. For example, a mutant with a mutator in a certain project can have a different effect on the source code and can give other results than a mutator in a different project. The relations between the mutants are scoped to their own project. In other words if a specific mutant has a relation to another mutant in the same project, this relation may not exist between the same mutant and a mutant from a different project. To preserve these relations and context we decided to train a separate model per project.
To validate the model we need to split our data set into a training set and a validation set. Research has shown that if the data set is big enough a ratio of 80/20 is sufficient[56]. We divide each source projects into training and validation groups to reflect the 80/20 ratio as best as possible. Depending on the amount of mutants per project we might not achieve a perfect 80/20 ratio. The division of the mutants is be done randomly. To make our results reproducible we select a random division based on a generated seed. The seeds generated for each run will be included in the results and can be found in the Github repository in the file *hierarchical-clustering/main.py* at line 182[57].

5.3 Implementation of algorithm

The research paper of the FCM algorithm is referenced in the survey of Du et al.[30]. The FCM algorithm is developed by Bezdek et al.[58]. Based on their research an implementation can be developed. There are existing libraries that contain the implementation of the FCM algorithm. Developing our own version of the FCM algorithm would be valuable if we want to make adjustments to the algorithm itself. For this experiment we do not require to make adjustments to the FCM algorithm. Therefore we choose to use a library that implements the FCM algorithm. The library we will use for this experiment is the fuzzy c-means implementation build by Dias et al.[59]. The theory developed by Bezdek et al., is implemented in this library. The library is actively maintained and has been used before in other research[60–62].

5.4 Algorithm parameters

The FCM algorithm has to initialize the centroids of the clusters. The library we selected does this based on a random number. It also allows us to control this random number. To make the initialization of the centroid reproducible we will use the same generated seeds as in our first experiment. The seeds generated are included in the Github repository in the file *hierarchical-clustering/main.py* at line 182[57]. Bezdek et al., also researched cluster validity for the FCM model[63]. They researched what effect which parameters could have on the validity of the clusters. They specifically analyzed the role of weighting exponent m(fuzziness parameter). They concluded that the best choice for m is in the interval [1.5, 2.5], whose mean and midpoint is m=2. Wu proposed a new guideline for selecting parameter m[64]. His point of view was that a large m value will make the fuzzy c-means more robust to noise and outliers. He suggests implementing fuzzy c-means with m=[1.5, 4]. When the data set contains noise and outliers, the fuzzifier m=4 is recommended in a theoretical upper bound case. Since our data set does not contain noise(all mutants consists of every characteristic) we choose to select the midpoint concluded in the research of Bezdek et al.[63]. The value of m in this experiment is 2.
Finding the correct amount of clusters has been an ongoing problem for clustering algorithms[30]. Fortunately there are techniques available to estimate the number of clusters. During model selection we discussed the mountain and subtractive models. These models are designed to estimate the number of clusters in a data set. The subtractive clustering model is specifically designed to approximate the number of clusters for fuzzy system models[54]. It approximates the centroids and amount of the clusters based on the data points in the data set[54]. We select and implement the subtractive clustering model to estimate the number of clusters. We use this number as the value for the number of clusters parameter in the FCM algorithm.

5.5 Implementation of subtractive clustering

For the same reasons as in Chapter 5.3 we select a library that implements the subtractive clustering algorithm. The research paper of the mountain and subtractive algorithms are referenced in the survey paper of Du et al.[30]. Based on the paper of the theory of the subtractive algorithm[54] we found an existing implementation[65]. The implementation is professionally build and maintained by Matlab[65]. Matlab is a tool designed for the academic world and is professionally maintained for this reason we decided to use their implementation of subtractive clustering[66].

Chapter 6

Experiment design

In this chapter we design an experiment to validate our research. We execute the following steps:

1. Gather all mutant characteristics from PIT.
2. Execute full set of mutants to gather baseline measurement.
3. Cluster the mutants according the black and white box methodologies.
4. Execute one random mutant per cluster and gather result.
5. Calculate weighted mutation score.

Both research questions have the same goal, but with different approaches. We can devise one hypothesis for both research questions. We hypothesise that we can cluster mutants with the set of characteristics we identified while maintaining effectiveness and reducing the amount executed when executing one mutant from each cluster that is randomly selected. Our goal is to achieve a weighted mutation score that is as close as possible to the mutation score of a full set of executed mutants. We repeat the experiment for each approach we devised in Chapters 4 and 5.

6.1 Project selection

We choose three main requirements for selecting software projects; the projects should have a test suite, the test suite should not contain failing tests and the mutation testing tool should be able to execute mutants for the sample project. We selected projects that were also used in other research within the mutation testing and testing domain[10, 14, 17, 34, 40, 67].
We extend our sample by selecting projects from the first six pages of the most popular Java projects on GitHub[1]. The unfiltered sample contains 50+ projects. From there we filter out all the projects that contain failing tests and that are not libraries or applications. The result consists of a sample with fifteen projects in total.

6.2 Baseline measurement

The baseline for our experiment is the mutation score of the set of all mutants generated by the selected mutation testing tool. To extract this score we need to execute a full set of mutants within PIT. For each source we have added the PIT plugin and required configuration in the *pom.xml*, this code instructs PIT to generate and execute mutants with all mutators available, an example of such code can be found in Appendix A. We can then use the Command Line Interface (CLI) and a maven command(see Listing 6.1) to execute PIT.

```
mvn -U org.pitest:pitest-maven:mutationCoverage
```

Listing 6.1: Command used to execute full set of mutants with PIT.

[1] https://github.com/search?p=7&q=language%3Ajava+stars%3A%3E10000&type=Repositories

6.3 Weighted mutation score

Our hypothesis states that each mutant executed should represent that whole cluster. With a mutant executed from each cluster we can calculate a mutation score. This mutation score is a weighted mutation score. This weighted mutation score is the product of the result of a mutant(1 for killed and 0 for survived) and the amount of mutants in the cluster it represents. The weighted mutation score is then comparable to the score of a full set as the total number of mutants will be the same.
For example, take a full set with a score of 75/100 killed mutants. This gives us a mutation score of 75%. We then cluster the mutants in four clusters consisting of 12, 30, 38 and 20 mutants, respectively. We randomly select four mutants of each cluster and execute them. The mutants representing cluster one and four survive and two and three are killed. If we calculate the weighted score we get 68/100 which is 68%. We can then compare this to the score of a full set because the amount of mutants executed is the same: 75/100(75%) and 68/100(68%).

6.4 Validation

The most efficient way to measure test effectiveness with mutation testing is by executing all mutants that a mutation testing tool can possibly generate. The goal of this research is to reduce the amount of mutants executed while maintaining effectiveness. To reduce the amounts executed we cluster the mutants. We can measure this by counting the number of clusters we generate and compare it to the number of total mutants generated by the selected tool.
To validate how effective our method of clustering is we can compare the weighted mutation score(see Chapter 6.3) of the clustered set to the mutation score of the full set. The closer the weighted score is to that of a full set the more effective our set of characteristics and clustering algorithm proves to be. In other words we want achieve a mutation score that is as close as possible as to that of a full set. We select a statistical significance level of ≤ 0.05. This is the conventional threshold for declaring statistical significance[68].
Depending on the effectiveness of our clustering algorithm we may loose accuracy. This can happen if a cluster contains mutants of both results. We can measure the accuracy inside a cluster by calculating a percentage of all mutants that have survived against the ones that have been killed in a cluster. If the majority of the mutants in a cluster is killed then we consider that cluster to represent a killed cluster and the other way around for survived mutants. We consider the mutants that are not in the majority of the cluster as inaccuracy.
If the weighted mutation score of our clustered set deviates more than 5% from the score of a full set we reject our hypothesis.

6.5 Random mutant selection

As stated in our hypothesis we randomly select a mutant from each cluster to be executed. To validate our sample we make use of statistical hypothesis testing[69]. Our null hypothesis states that there is no relation between the characteristics identified and the results of a mutant with the chosen methodology. To test this hypothesis we select an alpha value of 0.05. To translate; if more than five percent of the values in a sample deviates more than the significance level(see Chapter 6.4) we cannot reject the null hypothesis. To make our results reproducible we select the random mutant based on a generated seed. The seed generated for each run will be included in the results and can be found in the Github repository in the file *hierarchical-clustering/main.py* at line 182[57]. We repeat our experiment 30 times with 30 different seeds. Achieving consistent results while applying random selection contributes to the validity of the experiment.

Chapter 7

Results

In this chapter we present the results of our experiments. We divide the results in two sections. The first section displays the results of clustering with hierarchical clustering. The second section displays the results of clustering with the FCM model.
During the process of extracting the characteristics we found that calculating the Levenshtein distance took a long time. Calculating the Levenshtein distance for the four biggest projects in the data set cost days. Even for the smaller projects the extraction of the characteristics cost hours. Including the Levenshtein distance defeats the purpose of increasing performance as every bit of performance we increase would be replaced by the cost of calculating the Levenshtein distance. To see if it is possible to mitigate this risk we repeat the experiment without the Levensthein distance characteristic.

7.1 Hierarchical clustering

In the this section we present the results of the experiment with hierarchical clustering. We divide the results in two variants. The first variant displays the results of clustering with all characteristics. The second variant displays the results of clustering without the Levensthein distance.
For each variant and project we display the weighted mutation score to create box-plots. There are a total of eleven projects and thus also eleven box-plots per variant. In a box-plot the box is from the first quartile till the third quartile and a line at the median. The dots are measurement points laying outside the fences, which are located at Q1 1.5(Q3 Q1) and Q3+1.5(Q3 Q1). The whiskers show the range of points that are outside the box but inside the fences.
Due to the limitation of resources and time we had to omit four source projects from these results. The hardware was not equipped to store the amount of generated mutants for these projects, we did not have enough memory. Trying to cluster these mutants this resulted in a out of memory error.
The results displayed are the average of the results of the 30 repetitions of the experiment. The result of each single repetition can be found in our Github repository in the folder *experiment_results*[57].

7.1.1 Clustering with all characteristics

This section displays the results of our clustering algorithm when executed with data that consists of all characteristics. The results are be grouped based on the percentage in reduction of mutants executed. We also display the accuracy of the clusters per reduction. The results displayed are the average of the results of the 30 repetitions of the experiment. The result of each single repetition can be found in our Github repository in the folder *experiment_results*[57].

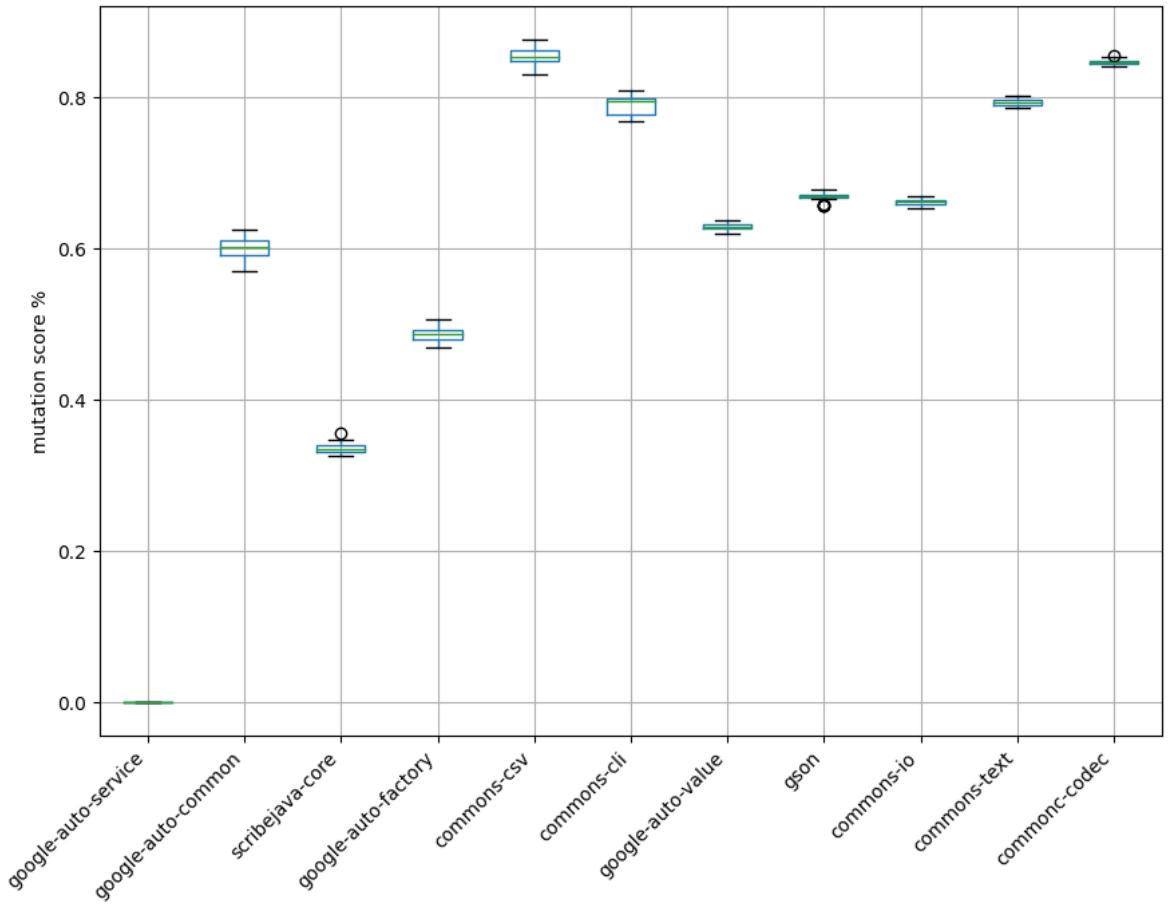


Figure 7.1: Box-plots containing weighted mutation score of clustering mutants with all characteristics and cluster size of n*0.25

Project	Mutation score full set	Mutation score clustered set	Avg. cluster accuracy	Min. cluster accuracy	Max. cluster accuracy
Google Auto Service	0.00%	0.00%	0.00%	0.00%	0.00%
ScribeJava-Core	33.52%	33.63%	93.29%	50.00%	100.00%
Google Auto Factory	48.41%	48.58%	94.18%	50.00%	100.00%
Google Auto Common	59.59%	60.14%	91.12%	50.00%	100.00%
Google Auto Value	61.42%	62.89%	94.33%	50.00%	100.00%
Google Gson	66.83%	66.87%	91.61%	50.00%	100.00%
commons-io	69.70%	66.18%	90.36%	50.00%	100.00%
commons-cli	79.07%	79.01%	91.35%	50.00%	100.00%
commons-text	79.29%	79.33%	89.95%	50.00%	100.00%
commons-codec	84.56%	84.71%	94.44%	50.00%	100.00%
commons-csv	85.16%	85.42%	92.08%	50.00%	100.00%

Table 7.1: Results of clustering mutants with all characteristics and cluster size of n*0.25

Figure 7.1 displays the box-plots of the weighted mutation score obtained from each individual sample. We can observe that for every box-plot the *p-value* is below 0.05 or 5%. To view the box-plots separately and in more detail see Appendix B.1. Table 7.1 displays the results of clustering mutants with all characteristics and the number of clusters equal to the total amount of mutants* 0.25. The maximum and minimum differences between the score of a full set and clustered are is 3.52% and 0.04% respectively. The average of the differences between full set score and clustered score is 0.58%. It is also noticeable

that the minimum and maximum accuracy are 50% and 100% respectively. This means that there was at least one cluster that consisted of only mutants that were killed or survived. The minimum accuracy cannot go below 50%. If the percentage does go below 50% the representation of a cluster will flip from survived to killed or the other way around.

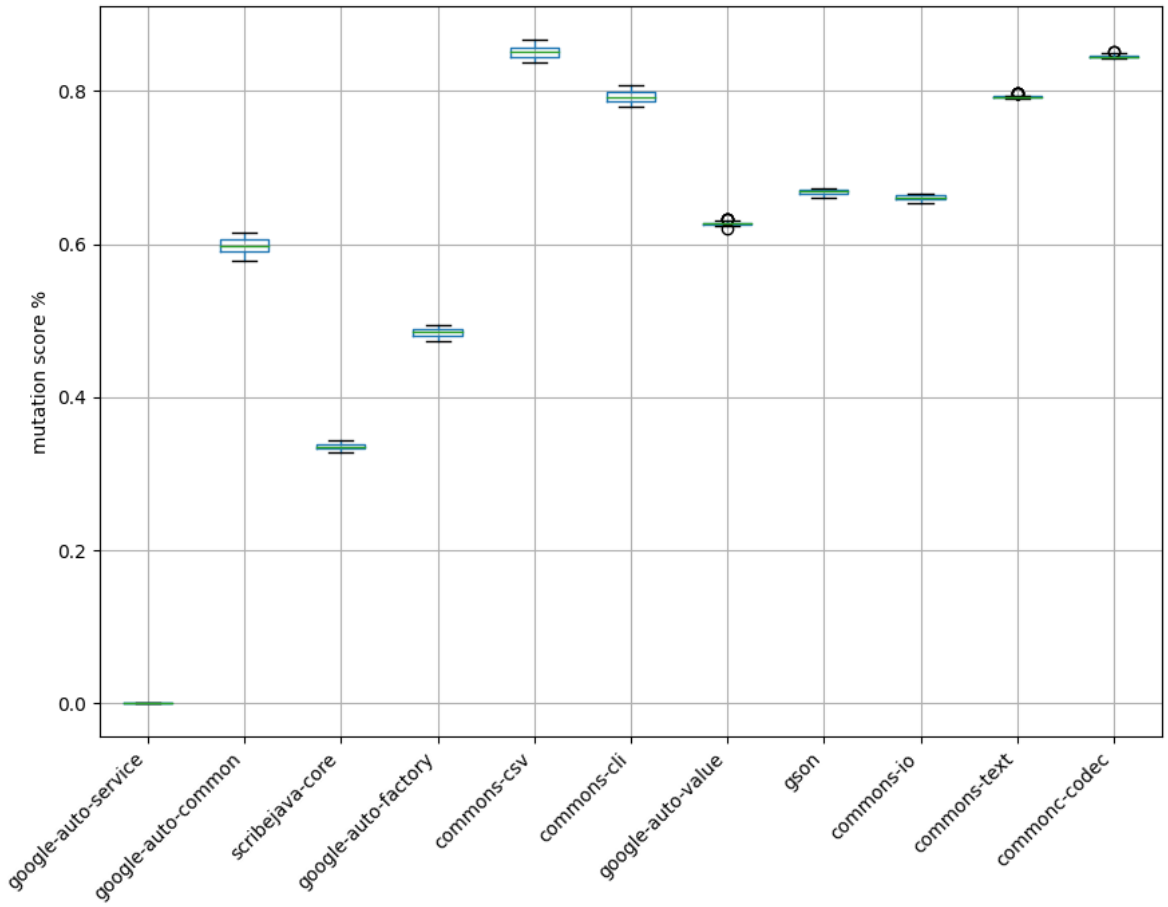


Figure 7.2: Box-plots containing weighted mutation score of clustering mutants with all characteristics and cluster size of n*0.5

Project	Mutation score full set	Mutation score clustered set	Avg. cluster accuracy	Min. cluster accuracy	Max. cluster accuracy
Google Auto Service	0.00%	0.00%	0.00%	0.00%	0.00%
ScribeJava-Core	33.52%	33.54%	96.94%	50.00%	100.00%
Google Auto Factory	48.41%	48.39%	96.12%	50.00%	100.00%
Google Auto Common	59.59%	59.76%	94.49%	50.00%	100.00%
Google Auto Value	61.42%	62.71%	96.09%	50.00%	100.00%
Google Gson	66.83%	66.81%	94.75%	50.00%	100.00%
commons-io	69.70%	66.06%	94.03%	50.00%	100.00%
commons-cli	79.07%	79.17%	94.19%	50.00%	100.00%
commons-text	79.29%	79.29%	93.21%	50.00%	100.00%
commons-codec	84.56%	84.60%	95.92%	50.00%	100.00%
commons-csv	85.16%	85.13%	94.65%	50.00%	100.00%

Table 7.2: Results of clustering mutants with all characteristics and size n*0.50

Figure 7.2 displays the box-plots of the weighted mutation score obtained from each individual sample.

We can observe that for every box-plot the *p-value* is below 0.05 or 5%. To view the box-plots separately and in more detail see Appendix B.2. Table 7.2 displays the results of clustering mutants with all characteristics and the number of clusters equal to the total amount of mutants* 0.50. The maximum and minimum differences between the score of a full set and clustered set are 3.63% and 0.02% respectively. The average of the differences between full set score and clustered score is 0.48% which is smaller than the average difference than that of the 25% set.

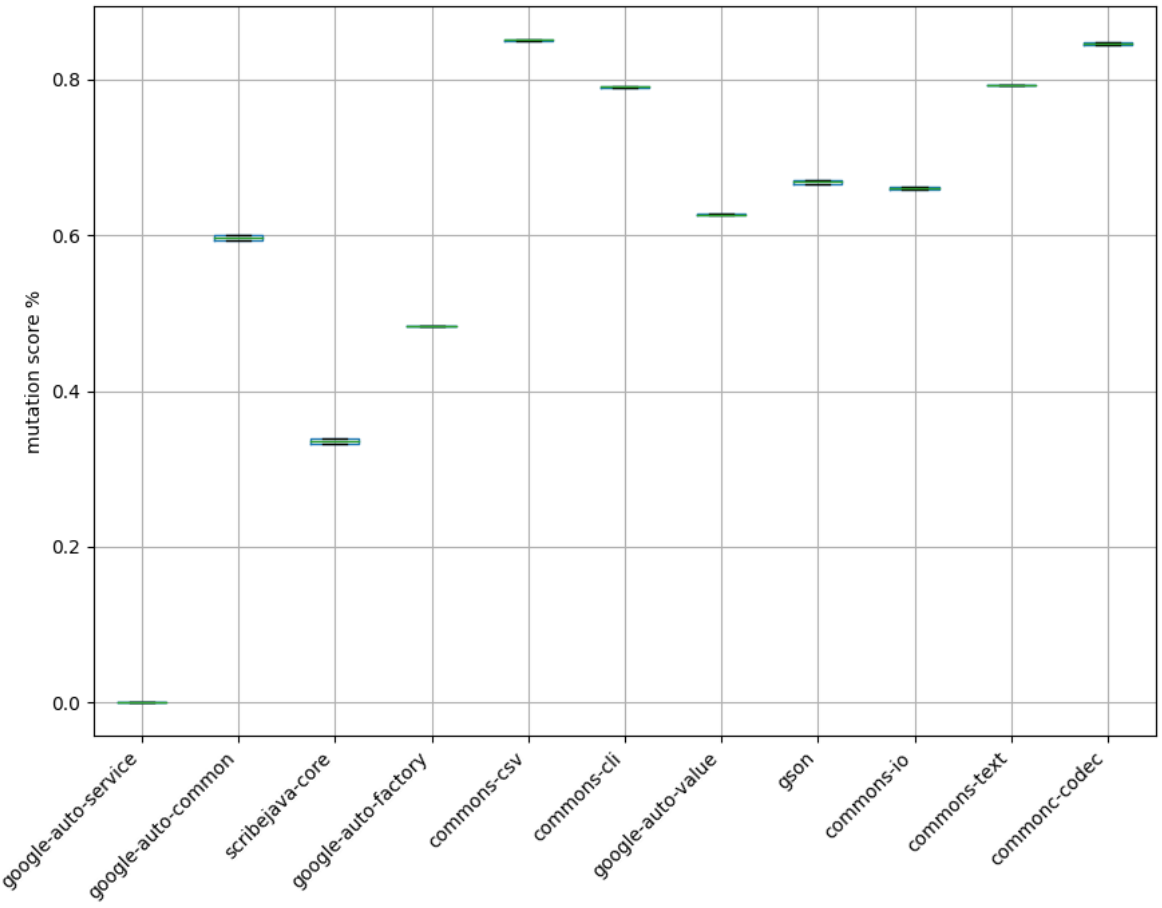


Figure 7.3: Box-plots containing weighted mutation score of clustering mutants with all characteristics and cluster size of n*0.75

Project	Mutation score full set	Mutation score clustered set	Avg. cluster accuracy	Min. cluster accuracy	Max. cluster accuracy
Google Auto Service	0.00%	0.00%	0.00%	0.00%	0.00%
ScribeJava-Core	33.52%	33.52%	97.99%	50.00%	100.00%
Google Auto Factory	48.41%	48.41%	97.46%	50.00%	100.00%
Google Auto Common	59.59%	59.76%	96.16%	50.00%	100.00%
Google Auto Value	61.42%	62.68%	97.31%	50.00%	100.00%
Google Gson	66.83%	66.84%	97.59%	50.00%	100.00%
commons-io	69.70%	66.07%	97.19%	50.00%	100.00%
commons-cli	79.07%	79.10%	96.68%	50.00%	100.00%
commons-text	79.29%	79.33%	97.61%	50.00%	100.00%
commons-codec	84.56%	84.61%	98.84%	50.00%	100.00%
commons-csv	85.16%	85.07%	97.64%	50.00%	100.00%

Table 7.3: Results of clustering mutants with a single characteristic and size n*0.75

Figure 7.3 displays the box-plots of the weighted mutation score obtained from each individual sample. We can observe that for every box-plot the *p-value* is below 0.05 or 5%. To view the box-plots separately and in more detail see Appendix B.3. Table 7.3 displays the results of clustering mutants with all characteristics and the number of clusters equal to the total amount of mutants* 0.75. The maximum and minimum differences between the score of a full set and clustered set are 3.63% and 0% respectively. The average of the differences between full set score and clustered score is 0.48% which is higher than the average difference than that of the 25% and the same of that of the 50% set.

7.1.2 Clustering without Levenshtein distance

This section displays the results of our clustering algorithm when executed without the Levenshtein distance. The results are grouped based on the reduced percentage of mutants executed. We also display the accuracy of the clusters per reduction amount. The results displayed are the average of the results of the 30 repetitions of the experiment. The result of each single repetition can be found in our Github repository in the folder *experiment_results*[57].

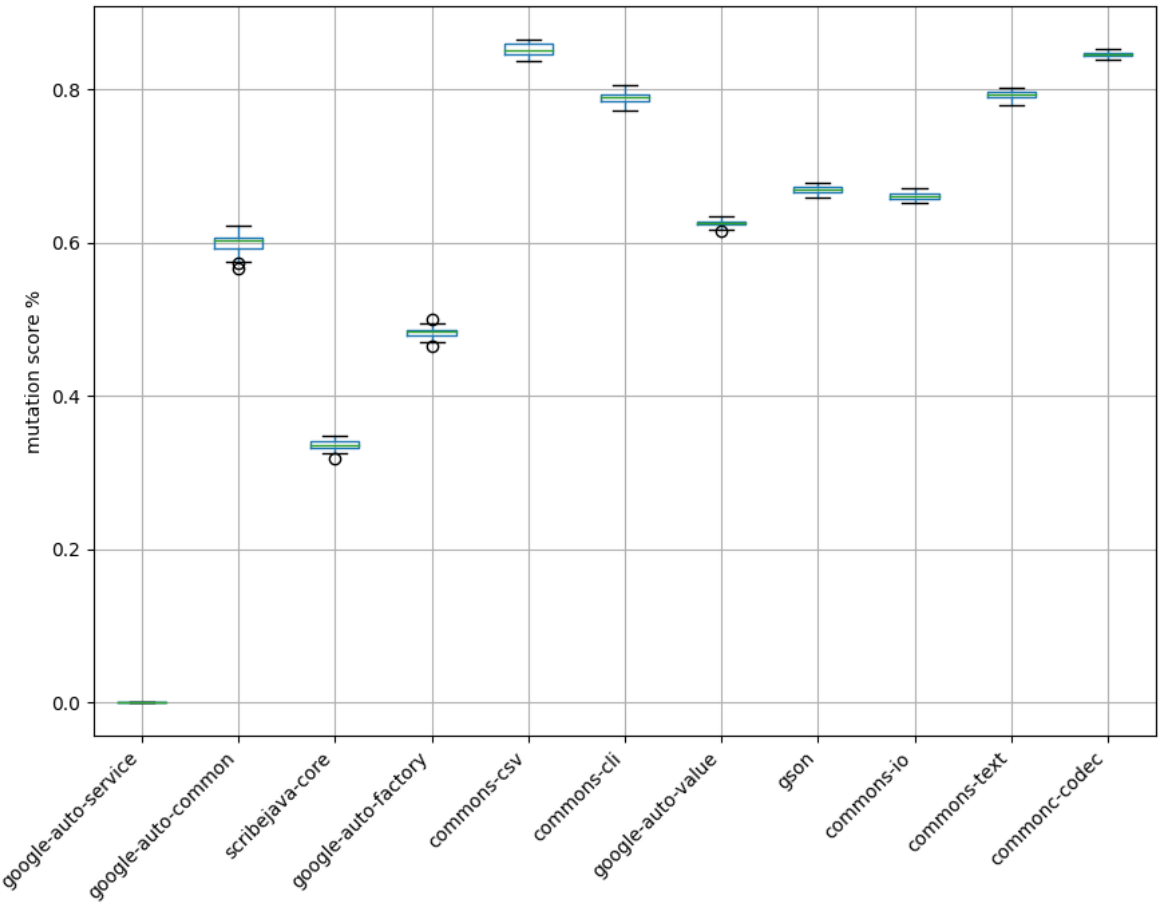


Figure 7.4: Box-plots containing weighted mutation score of clustering mutants without Levensthein distance and cluster size n*0.25

Project	Mutation score full set	Mutation score clustered set	Avg. cluster accuracy	Min. cluster accuracy	Max. cluster accuracy
Google Auto Service	0.00%	0.00%	0.00%	0.00%	0.00%
ScribeJava-Core	33.52%	33.72%	92.24%	50.00%	100.00%
Google Auto Factory	48.41%	48.32%	94.21%	50.00%	100.00%
Google Auto Common	59.59%	60.03%	91.19%	50.00%	100.00%
Google Auto Value	61.42%	62.58%	93.90%	50.00%	100.00%
Google Gson	66.83%	66.89%	91.75%	50.00%	100.00%
commons-io	69.70%	66.08%	90.20%	50.00%	100.00%
commons-cli	79.07%	78.97%	91.11%	50.00%	100.00%
commons-text	79.29%	79.30%	90.95%	50.00%	100.00%
commons-codec	84.56%	84.59%	94.56%	50.00%	100.00%
commons-csv	85.16%	85.24%	92.76%	50.00%	100.00%

Table 7.4: Results of clustering mutants without Levensthein distance and cluster size n*0.25

Figure 7.4 displays the box-plots of the weighted mutation score obtained from each individual sample. We can observe that for every box-plot the *p-value* is below 0.05 or 5%. To view the box-plots separately and in more detail see Appendix B.4. Table 7.4 displays the results of clustering mutants without Levensthein distance and the number of clusters equal to the total amount of mutants * 0.25. We observe that the maximum and minimum differences in mutation score between a full set and a clustered

set 3.61% and 0.09% respectively. The average of the differences between full set score and clustered score is 0.53%.

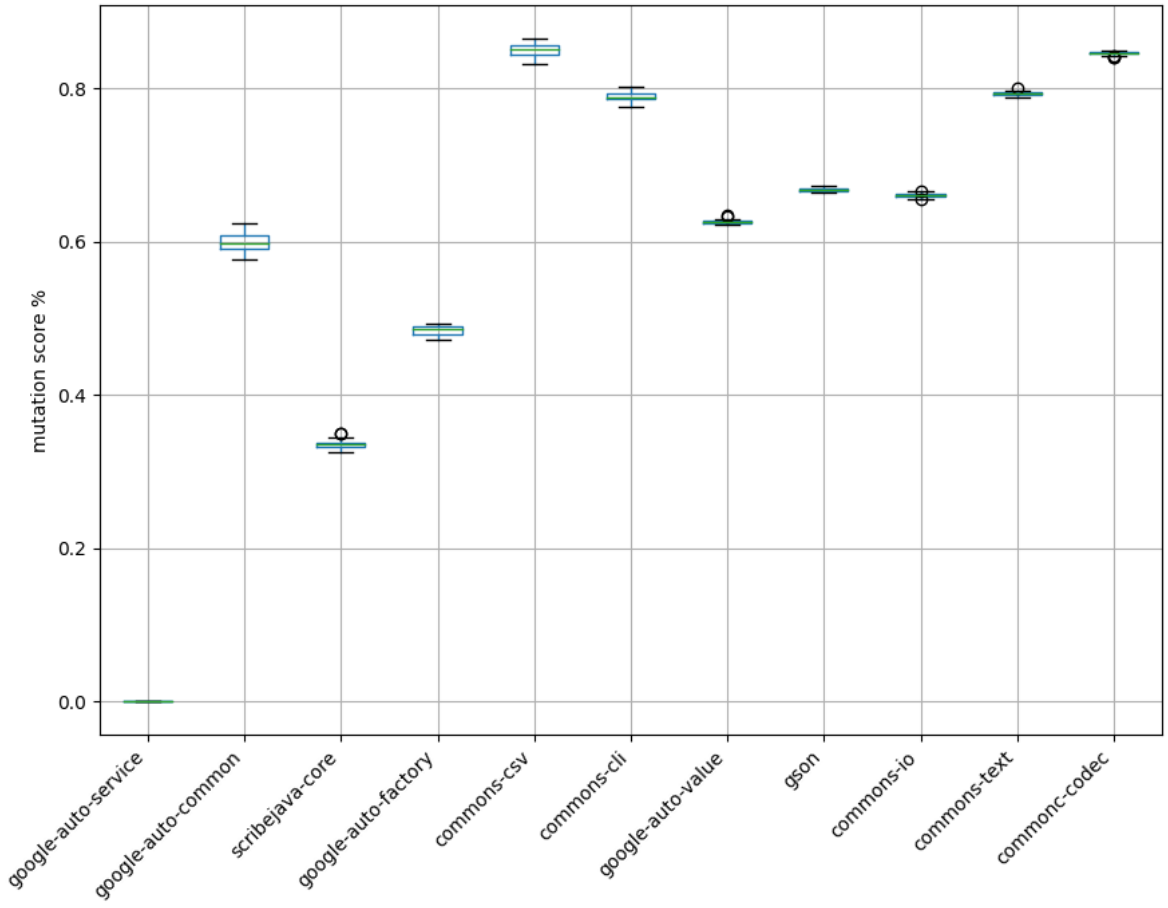


Figure 7.5: Box-plots containing weighted mutation score of clustering mutants without Levensthein distance and cluster size n*0.50

Project	Mutation score full set	Mutation score clustered set	Avg. cluster accuracy	Min. cluster accuracy	Max. cluster accuracy
Google Auto Service	0.00%	0.00%	0.00%	0.00%	0.00%
ScribeJava-Core	33.52%	33.55%	97.16%	50.00%	100.00%
Google Auto Factory	48.41%	48.44%	97.71%	50.00%	100.00%
Google Auto Common	59.59%	59.86%	96.35%	50.00%	100.00%
Google Auto Value	61.42%	62.62%	97.31%	50.00%	100.00%
Google Gson	66.83%	66.81%	95.29%	50.00%	100.00%
commons-io	69.70%	66.05%	95.08%	50.00%	100.00%
commons-cli	79.07%	78.97%	95.55%	50.00%	100.00%
commons-text	79.29%	79.30%	95.34%	50.00%	100.00%
commons-codec	84.56%	84.59%	96.95%	50.00%	100.00%
commons-csv	85.16%	85.11%	95.87%	50.00%	100.00%

Table 7.5: Results of clustering mutants without Levensthein distance and cluster size n*0.50

Figure 7.5 displays the box-plots of the weighted mutation score obtained from each individual sample. We can observe that for every box-plot the *p-value* is below 0.05 or 5%. To view the box-plots separately

and in more detail see Appendix B.5. Table 7.5 displays the results of clustering mutants without Levensthein distance and the number of clusters equal to the total amount of mutants * 0.50. The maximum and minimum differences between the score of a full set and clustered set are 3.65% and 0.03% respectively. We observe that the average cluster accuracy is higher than the average cluster accuracy of the 25% set. The average of the differences between full set score and clustered score is 0.49% which is smaller than the average difference than that of the 25% set.

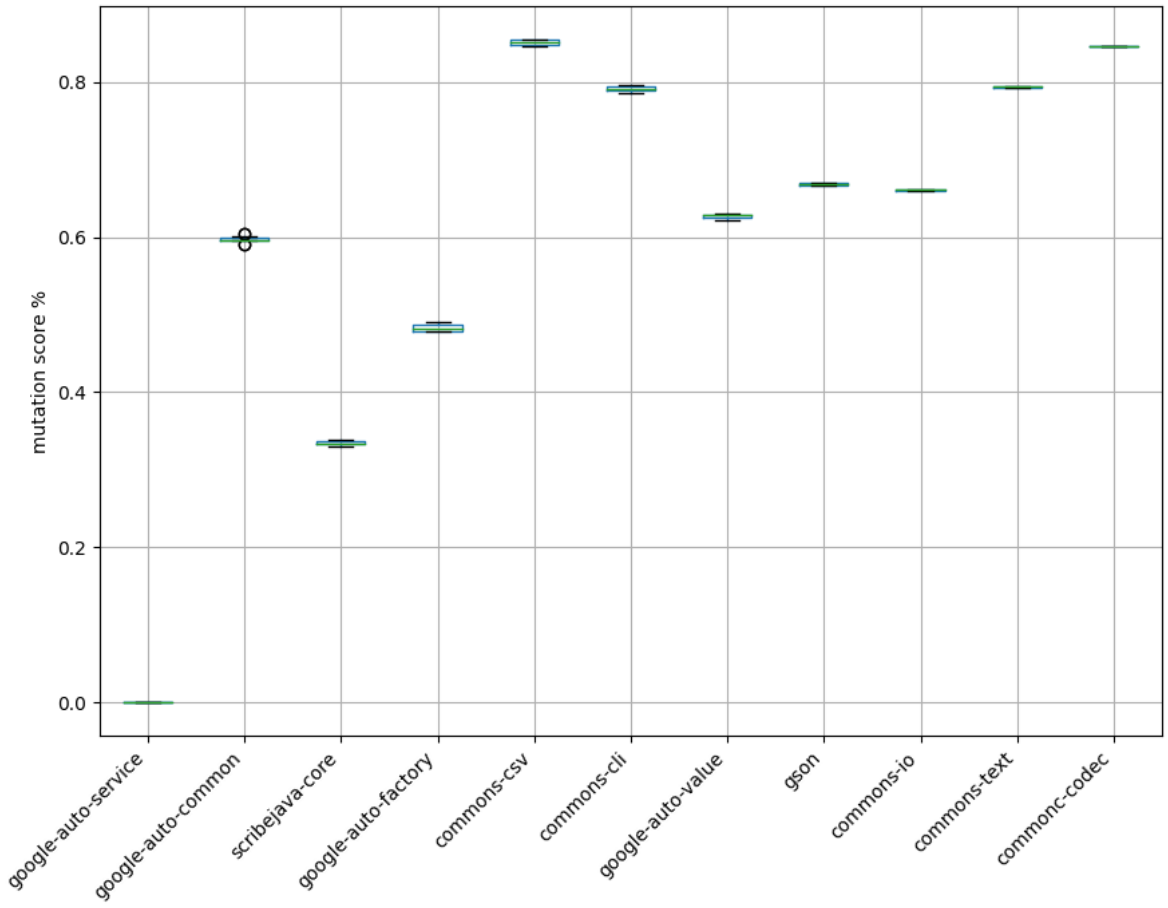


Figure 7.6: Box-plots containing weighted mutation score of clustering mutants without Levensthein distance and cluster size n*0.75

Project	Mutation score full set	Mutation score clustered set	Avg. cluster accuracy	Min. cluster accuracy	Max. cluster accuracy
Google Auto Service	0.00%	0.00%	0.00%	0.00%	0.00%
ScribeJava-Core	33.52%	33.48%	98.63%	50.00%	100.00%
Google Auto Factory	48.41%	48.34%	98.41%	50.00%	100.00%
Google Auto Common	59.59%	59.71%	97.08%	50.00%	100.00%
Google Auto Value	61.42%	62.66%	97.99%	50.00%	100.00%
Google Gson	66.83%	66.84%	97.37%	50.00%	100.00%
commons-io	69.70%	66.07%	97.34%	50.00%	100.00%
commons-cli	79.07%	79.12%	96.83%	50.00%	100.00%
commons-text	79.29%	79.33%	97.60%	50.00%	100.00%
commons-codec	84.56%	84.61%	98.43%	50.00%	100.00%
commons-csv	85.16%	85.07%	97.49%	50.00%	100.00%

Table 7.6: Results of clustering mutant without Levensthein distance and cluster size n*0.75

Figure 7.6 displays the box-plots of the weighted mutation score obtained from each individual sample. We can observe that for every box-plot the *p-value* is below 0.05 or 5%. To view the box-plots separately and in more detail see Appendix B.6. Table 7.6 displays the results of clustering mutants without Levensthein distance and the number of clusters equal to the total amount of mutants * 0.50. The maximum and minimum differences between the score of a full set and clustered set are 3.63% and 0.04% respectively. We observe that the average accuracy is higher than the average accuracy of the other sets. The average of the differences between full set score and clustered score is 0.49% which is higher than the average difference than that of the 25% and the same of that of the 50% set.

7.2 FCM model

In this chapter we present the results of our second experiment. We divide the results in two sections. For each variant and project we display the weighted mutation score to create box-plots. The first section displays the results of clustering with all characteristics. For the same reason as in Chapter 7.1 we repeat the experiment without the Levenshtein distance. The second section displays the results of clustering without the Levenshtein distance.
The results displayed are the average of the results of the 30 repetitions of the experiment. The result of each single repetition can be found in our Github repository in the folder *experiment_results*[57].

7.2.1 Clustering with all characteristics

This section displays the results of our clustering algorithm when executed with data that consists of all characteristics. Instead of displaying the reduction amount, which we did with the results of experiment one, we display the number of clusters that is calculated by the subtractive clustering model.

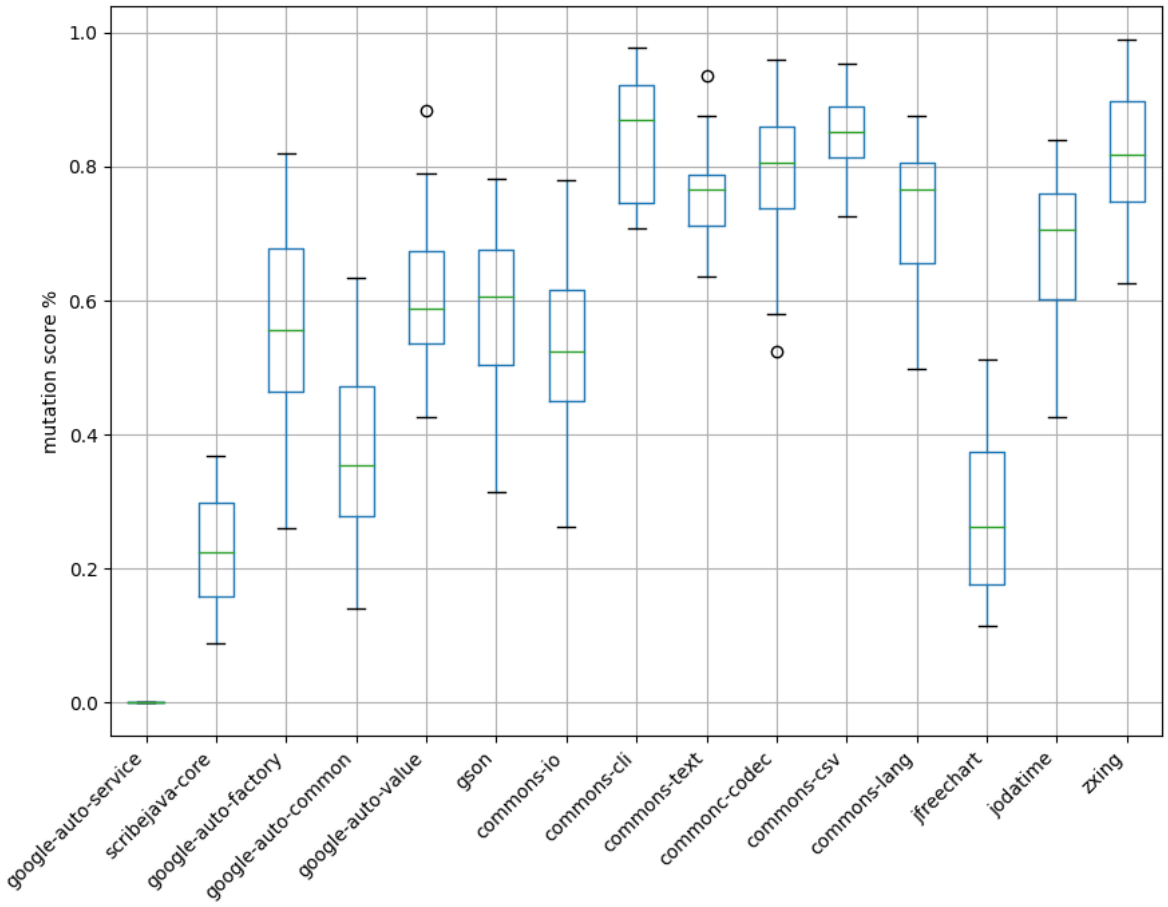


Figure 7.7: Box-plots containing weighted mutation score of clustering mutants with all characteristics.

Project	Number of clusters	Mutation score full set	Mutation score clustered set	Average cluster accuracy	Min. cluster accuracy	Max. cluster accuracy
Google Auto Service	17	0.00%	0.00%	0.00%	0.00%	0.00%
ScribeJava-Core	32	33.52%	22.91%	76.94%	59.60%	96.42%
Google Auto Factory	22	48.41%	56.59%	59.08%	50.60%	73.80%
Google Auto Common	31	59.59%	37.20%	63.04%	50.78%	87.93%
Google Auto Value	41	61.42%	61.01%	61.42%	50.94%	86.48%
Google Gson	40	66.83%	58.95%	59.22%	50.54%	79.57%
Commons-io	40	69.70%	53.32%	57.28%	50.63%	74.77%
Commons-cli	26	79.07%	84.08%	81.69%	67.87%	94.86%
Commons-text	55	79.29%	75.73%	78.19%	62.08%	93.54%
Commons-codec	17	84.56%	78.82%	77.67%	74.43%	81.08%
Commons-csv	32	85.16%	84.99%	83.41%	67.44%	96.39%
Commons-lang	28	78.31%	72.42%	77.04%	73.73%	81.19%
Jfreechart	28	27.02%	28.49%	74.08%	71.79%	76.30%
Jodatime	25	70.30%	67.51%	64.68%	60.73%	68.68%
Zxing	28	70.64%	81.50%	83.10%	80.20%	86.36%

Table 7.7: Results of clustering mutants with the FCM model with all characteristics

Figure 7.7 displays the box-plots of the weighted mutation score obtained from each individual sample.

We can observe that for every box-plot the *p-value* is more than 0.05 or 5%. Table 7.7 displays the results of clustering mutants with a all characteristics. The maximum and minimum differences between the score of a full set and clustered set are 22.93% and 0.4% respectively. The average of the differences between full set score and clustered score is 5.43%.

7.2.2 Clustering without Levenshtein distance

This section will display the results of our clustering algorithm when executed with data that does not contain the Levenshtein distance characteristic. Instead of displaying the reduction amount, as we did in the results of experiment one, we display the number of clusters that is calculated by the subtractive clustering model.

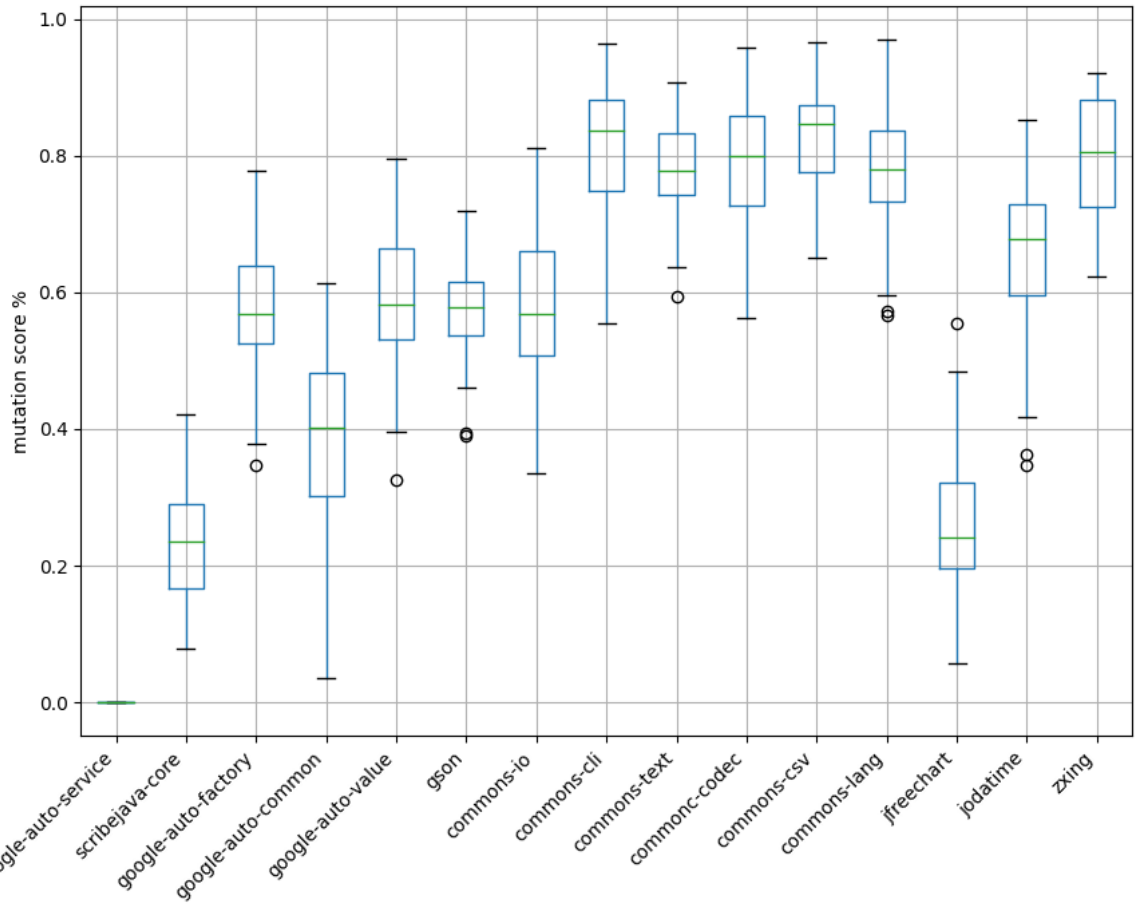


Figure 7.8: Box-plots containing weighted mutation score of clustering mutants without Levenshtein distance characteristic.

Project	Number of clusters	Mutation score full set	Mutation score clustered set	Average cluster accuracy	Min. cluster accuracy	Max. cluster accuracy
Google Auto Service	17	0.00%	0.00%	0.00%	0.00%	0.00%
ScribeJava-Core	32	33.52%	23.64%	76.80%	58.77%	96.60%
Google Auto Factory	22	48.41%	57.40%	59.20%	50.67%	73.96%
Google Auto Common	31	59.59%	37.64%	62.95%	50.59%	83.53%
Google Auto Value	41	61.42%	59.61%	61.48%	50.60%	83.53%
Google Gson	40	66.83%	57.61%	59.11%	50.84%	76.68%
Commons-io	40	69.70%	57.95%	57.25%	50.62%	72.70%
Commons-cli	26	79.07%	81.53%	81.77%	67.70%	95.75%
Commons-text	55	79.29%	77.92%	78.22%	61.80%	93.92%
Commons-codec	17	84.56%	79.46%	77.73%	74.39%	81.66%
Commons-csv	32	85.16%	82.84%	83.47%	65.58%	97.80%
Commons-lang	28	78.31%	76.83%	76.99%	73.03%	80.56%
Jfreechart	28	27.02%	26.41%	74.11%	71.83%	76.36%
Jodatime	25	70.30%	65.17%	64.75%	60.82%	69.81%
Zxing	28	70.64%	80.03%	83.11%	80.10%	86.03%

Table 7.8: Results of clustering mutants with the FCM model without Levenshtein distance

Figure 7.8 displays the box-plots of the weighted mutation score obtained from each individual sample. We can observe that for every box-plot the *p-value* is more than 0.05 or 5%. Table 7.8 displays the results of clustering mutants without Levenshtein distance. The maximum and minimum differences between the score of a full set and clustered set are 21.95% and 0.61% respectively. The average of the differences between full set score and clustered score is 4.44% which is bigger than the average difference than that of the results with all characteristics.

Chapter 8

Discussion

In this chapter, we discuss the results of our experiments on clustering mutants. First, we discuss the results of the white box approach. The results are discussed as one and are compared between the variants(with Levenshtein distance characteristic and without). Second we discuss the results of the black box approach and compare them to those of the white box approach. And last we discuss the practicality of our research.

8.1 RQ 1: Identifying and clustering mutant characteristics

The maximum deviation from the original mutation score, for all variants(with and without Levenshtein distance) of the experiment, is below the statistical threshold. We can observe that the distance between the whiskers of the box-plots do not stretch more than the *alpha-value* of 0.05. This means that there is not one sample that deviated more than 5% from the original mutation score. Thus we can reject the null hypothesis and accept our alternative hypothesis. As hypothesised we observe that it is possible to cluster mutants with the set of characteristics we identified while maintaining effectiveness and reducing the amount executed. The hypothesis holds for all the projects and variants of the experiment. 66% of the displayed results have a deviation of less than 1%. This includes both variants of the experiment.
The results prove that the relations between the mutants represented by the characteristics we identified can be used to cluster mutants efficiently. This can be explained by the correlation between the result of a mutant, the calculated euclidean distance and the distance threshold(linkage criteria) used to decide whether a mutants should be clustered together.
If two mutants are grouped it is n percent likely that the mutants have the same result where n is the cluster accuracy.

> **Finding 1:** When executing 25% of the total amount of mutants we can still maintain effectiveness.

In both variants of the experiment we observe a reduction in average accuracy when increasing the performance by reducing the number of clusters. These reductions are 0.1% and 0.04% for clustering with and without Levenshtein distance respectively. We can also see that the average cluster accuracy is lower when reducing the amount executed.

> **Finding 2:** The more you reduce the number of mutants executed the more accuracy is lost.

The results show that the average accuracy for reductions of 25% and 50% is the same. This observation shows true for both variants of the experiment which is remarkable.

> **Finding 3:** We can reduce the number of mutants executed in our data set up to 50% before losing more accuracy.

It is also remarkable to see the min and max accuracy of the all clusters are 50% to 100% respectively. Based on these results we can state the following:

- For every project there was at least one cluster that consisted only of killed or survived mutants.

- For every project there was at least one cluster that evenly divided between killed and survived mutants.

With the exception of Google Auto Service and Commons Text, a reduction of 25% and all characteristics, all projects showed an average accuracy above 90%. The combination of hierarchical clustering and the identified characteristics results in high accuracy clusters.
According the rules of logic, a higher cluster accuracy means a score closer to that of the original. In other words if all clusters had an accuracy of 100% we will achieve the same mutation score as that of the full set. However we do not observe this behaviour in our results. This is due to our random sampling method. If we would select a mutant in the majority of each cluster the accuracy per cluster would correlate to the deviation in mutation score. Unfortunately we do not know the majority of a cluster as we do now know if a mutant survives or is killed before execution.

Finding 4: Random sampling a mutant per cluster may result in a bigger deviation between original score and weighted score.

As stated in Chapter 7.1 the calculation of the Levenshtein distance characteristic took a long time. For this reason we repeated the experiment without the Levenshtein distance characteristic. Comparing the results between these variants we can observe a max deviation of 0.05% between average difference of the mutation scores. In 66% of the results displayed in Chapter 7.1 clustering without the Levenshtein distance is more accurate. The biggest difference in mutation score measured between all the projects and their reductions is 3.63%. Clustering without Levenshtein distance characteristic is on average more accurate.

Finding 5: While clustering without the Levenshtein distance characteristic was more accurate in our results, the Levenshtein distance characteristic does not significantly increase or decrease the cluster accuracy.

8.2 RQ 2: Training a machine learning model and clustering mutant characteristics

The maximum deviation from the original mutation score, for all variants(with and without Levenshtein distance characteristic) of the experiment, is above the statistical threshold. We can observe that the distance between whiskers of the box-plots is more than the *alpha-value* of 0.05. Thus we cannot reject the null hypothesis and accept our alternative hypothesis. The biggest deviation measured is 22.39%. This is significantly higher than the biggest deviation measured in the first experiment. 56% of the results of both variants had a deviation in mutation score bigger than 5%.

Finding 6: Our hypothesis does not hold for the clustering the mutants with the FCM model trained in our experiment.

We observe that the number of clusters does not have an obvious relation to the total amount of mutants. This is expected as the subtractive clustering does not look at the total amount of data points(mutants) but to the contents(characteristics) of the data points. The number of clusters compared to the that of the hierarchical clustering is significantly lower. While we do not know the exact clusters centroids the subtractive model calculated, a low number of clusters indicates close relationships between certain mutants.

Finding 7: The mutants represented as characteristics in our data set per project have, according to the subtractive clustering model, close relationships to certain other mutants in the same data set.

We observe that the minimum lowest minimum cluster accuracy measured over all variants and projects is 50.54% with and average of 62%. This is higher than the minimum of the hierarchical clustering. This means that in all project there was not one cluster that was evenly divided. While the minimum accuracy was higher, the maximum accuracy was lower compared to the maximum accuracy of the hierarchical clustering results. The highest accuracy measured over all variants and projects is

97.80% with an average of 83.79% This means that in all project there was not one cluster that consisted of only survived or killed mutants.

Finding 8: The overall accuracy of the clusters generated by the FCM model in both variants is lower compared to that of the hierarchical clustering.

When we compare the accuracy per project between the FCM model and hierarchical clustering we can observe that the lower accuracy of the FCM model clusters have a higher impact on the deviation in mutation score. This is caused by the relatively low number of clusters. One cluster contains more mutants which results in a bigger weight per cluster.

Finding 9: The inaccuracy of clusters has a bigger impact on mutation score when the number of clusters is lower.

8.3 Application

The application of our research is limited to all software projects written in the language Java and the requirements of PIT[39]. We can repeat our experiment for every java project that can be run by PIT. The scope of the research done in the domain of mutation clustering(Chapter 9) is a subset of the scope in our research. With our research it is possible to cluster all the mutants used in other related research. The extraction of mutants is build within the PIT plugin system which is a plug and play system. Applying our research can be done in three steps. The first step is to run PIT with the plugin to gather the characteristics. Gathering the characteristics does not take more than a few minutes depending on the number of mutants generated. The plugin also makes sure PIT exits without executing mutants after gathering the characteristics. The second step is to run the script for either the FCM model or the hierarchical clustering. The last step is to run PIT again with the plugin to filter the mutants based on the results of the clustering. It is possible to build all these steps into a DevOps pipeline to automate the process. The plugins and scripts needed for these steps are available in the code repository on Github[57]. The plugins can be found in the directory *pitest-clustering-plugin*. The scripts used can be found in the directories *FCM-clustering* and *hierarchical-clustering*.

8.4 Threats to validity

In this section, we discuss the threats to internal validity and external validity.

8.4.1 Internal validity

Equivalent mutants

Equivalent mutants are mutants that are syntactically different but functionally identical. These mutants influence the mutation score while giving the same result as their equivalent counterpart. In our research equivalent mutants could give a false image of the accuracy of the clusters and the overall accuracy if such a mutant was randomly selected. PIT contains some functionality for reducing equivalent mutants[39]. We also reduced the chance of inaccuracy by repeating our experiment.

Test set representativity

The software projects we used in our research were systematically selected. Chapter 6.1 provides an indication but not a wide representation of industrial software projects. The conclusions drawn regarding the increase in performance by reducing the mutants are based on the comparison between our own baseline measurements.
Another threat which is the result of the representation of our test set is the representation of the mutants. While our data set contains around 1 million mutants we do not know if we have seen all mutants that exist. This will always be a problem as it would be nearly impossible to cluster and mutation test every piece of Java code, that conforms to the requirements of our research, ever written.

Project configuration

Some tests may not be recognized by PIT due to some external data required for the test. Another reason may be that the tests uses global variables that are not available to PIT. We encountered some of these tests in our data set. While these tests did pass when run with only the unit test framework they would fail when run by PIT. These projects still met the requirements stated in Chapter 6.1. To solve this issue we removed these tests from the test suite. By removing these specific tests we may have slightly, tough negligible, deviated from the individual real-world settings for respective projects.

Expertise

While this research is in the domain of software engineering it borrows techniques from data science. The researcher's field of expertise is software engineering. Since the researcher had to use techniques from different domains we can only guarantee that the research executed was done to the best of their ability. Thus the research may contain sub optimal design choices which may have been made from a software engineering perspective.

8.4.2 External validity

We clustered mutants with two different techniques. Our results are not generalisable to projects written in other programming languages as well as projects that cannot be run by PIT. As a result of using PIT we limit ourselves to two testing frameworks; JUnit and TestNG. PIT is still under active development and may expand its applicability, which in turn increases the applicability of this research.

Chapter 9

Related work

Clustering mutants has been done before. The research that exists tries to cluster mutants by defining the centroid of a cluster and cluster mutants on that definition. Our research extends mutation clustering by defining a centroid that is dynamic. In other words the centroids of our clusters are defined by the mutants itself.

9.1 Clustering overlapped mutants

Ma et al., clustered mutants by overlap[13]. They defined the term overlap as as mutants that are functionally equivalent. This is close to an equivalent mutant but not the same. They explain that an equivalent mutants is functionally identical to the original source code while an overlapped mutant is functionally identical to at least one other mutant. If a mutant does not have an overlapped mutant that mutant would be a cluster on its own. The achieved a reduction, in mutants executed, of 10%. They write in their research that it is limited to the mutator operators on expression level. If a mutator did not generate two or more mutants they could not detect overlapping mutants. Their future work includes adding more mutators and widening the scope level to statements or blocks.
The research does not include efficiency. While they show that they can cluster mutants based on overlap they do not show how efficient the mutation testing process is compared to executing a full set of mutants. A threat to their validity, which is not addressed, is the fact that two overlapping mutants may not be challenged by the same tests and thus can result in different results. Our research solves this problem by taking into account the location of the mutants. Chapter 4.1.7 describes what characteristics we use.

9.2 Clustering by scope

Yu et al., extended the research about overlapping mutants(see chapter 9.1)[14]. They extended the scope by adding more mutators. The also extended the research to include statements and blocks. Their results shows an increase in clustered mutants. The research of Yu et al., suffers from the same limitations and threats mentioned in Chapter 9.1.

9.3 Clustering Hamming distance

Ji et al., did a qualitative domain analysis on mutants[15]. As a result they identified the Hamming distance to cluster with. The Hamming distance is, like the Levenshtein distance, a similarity measure. They use the k-means algorithm to cluster mutants represented by Hamming distance. They write that the reduced test set in their experiment is still as strong as the original test set[15].
Ji et al., were successful in clustering mutants with Hamming distance and the k-means algorithm. They do acknowledge that their research still has problems with rationally determining the domains of their clusters. Our research does not have this problem, as the centre of the domains are decided by the characteristics of the mutants.

9.4 Spectral clustering

Wei et al., makes use of an intelligent technique, namely spectral clustering, to improve the efficacy of mutant reduction[17]. They defined multiple definitions for the mutants; similarity distance, distance between the mutants and a killing matrix of the mutants. With these definitions they reduced the mutants based on their proposed reduction method. This method is based on spectral clustering (SCMT), the determination method of the number of clusters, spectral clustering of mutants, and selection of representative mutants. The reduced mutants were then clustered with a classical clustering algorithm. Their results are promising and show high cluster accuracy. They write that there is still work left to do in optimizing the matrices and clustering algorithms. Our goal overlaps with that of Wei, the difference is that we used a different methodology to achieve that goal. Another difference is that our research does not require to know the result of a mutant(killed or survived).

9.5 Clustering similarity

Hussain et al., used the k-means and agglomerative clustering algorithm to cluster mutants according to a similarity measure. They used the Hamming distance as similarity measure. They calculated the distance and used this as data to feed into the clustering algorithms. The number of clusters and the initial position of the cluster center in the k-means algorithm are difficult to determine, and the process of the agglomerative clustering algorithm makes it difficult to correct the existing cluster formation[70]. They did take into account efficiency. Hussain et al, had the same goals as our research. The difference is that our research used different and more elaborated methods to cluster mutants. Hussain et al., calculated the mutation score by counting one cluster as one mutant. Our research calculated a weighted mutation score to reflect the clusters more accurately.

9.6 Generalizing mutants

Wilinski et al., tried to generalize the mutants by defining metrics. Each metric is calculated for a specific mutator. The three metrics are usefulness, frequency and dependency. Combining the results of usefulness and frequency metrics, they observed that reducing the number of generated mutants gives noticeable cost reduction without a loss of the mutation score accuracy. Wilinski et al., their research is narrowly scoped to the specific mutators they decided to do research on. Our research does not contain this limitation. The mutator is used as a characteristic in our research instead of limiting our research it helps define a mutant.

Chapter 10

Conclusion

Our research started with comparing three mutation testing tools for Java and found that the state of the art was outdated. We replicated the state of the art research with current data available on the mutation testing tools. The state of art showed that two out of three mutation testing tools were not actively maintained anymore. The mutation testing tool PIT scored best on the metrics measured in our replication study, this is also the tool we decided to use for our research.
We performed a qualitative analysis on mutant characteristics. As a result we created a list of identified characteristics based on existing research. We then extended this list by identifying characteristic with logic and data available. We used the identified characteristics to represent the mutants.
We clustered the mutants with a white box approach and a black box approach. For our white box approach we selected the hierarchical clustering algorithm. The hierarchical clustering showed that we can cluster mutants while maintaining effectiveness. We achieved reductions of up to 75% of the total amount of mutants without significantly reducing the accuracy.
For the black box approach we selected a machine learning model, specifically the fuzzy c-means model. The FCM model showed a significant reduction in effectiveness. We trained theFCM model with parameters that were optimal according to existing research. We used a different machine learning model, specialized in calculating the number of clusters for fuzzy clustering, to calculate the number of clusters. While the minimum clustering accuracy was higher than that of the hierarchical clustering, the maximum cluster accuracy was lower than that of the hierarchical clustering. The cluster accuracy had a larger impact on the clusters of the machine learning model because there was a relatively small number of clusters. Our research proves that it is possible to determine the means of domains as the centre of a cluster within a programming language without narrowing down the scope to specific pieces of code. The centroids our research identified is decided by the characteristics of the mutants. It can be applied to all Java projects that can be run with PIT. Which is a larger scope than in existing mutation clustering research.

10.1 Future work

Optimizing parameters of machine learning model

We trained the machine learning model with specific parameters guided by relevant research. These proved to be non optimal for our purpose. This research can be extended by finding the optimal parameters for the SOM clustering algorithm. The results showed a higher minimum accuracy which may indicate that there is potential for clustering the mutants with the machine learning model. Further research and fine-tuning is required to find out if the model can be trained to achieve more accurate scores.

Training strategy

Our approach for training the model proved to be non optimal. During the training of the model we experienced that the more clusters you select the more memory is needed for training. This research did not have sufficient hardware available. A different approach in training strategy may be explored to mitigate the hardware limitation.

Training a single machine learning model

For this researched we trained a model for each source project. A different way is to train a machine learning model with the combined set of mutants. The model could be continuously trained each time a new source is added. Further research is recommended in this direction as it will improve the applicability of this research.

Different sets of characteristics

We have proven that this set of characteristics can achieve accurate results in combination with hierarchical clustering. It would be valuable to find out if it possible to reduce or expand the set of characteristics and still maintain accurate results. Further experimentation may be done with different sets of characteristics.

Acknowledgements

I would like to thank my supervisor, dr. Ana Oprescu, for her guidance ever since my first semester on the University of Amsterdam. Thank you Ana, for being my supervisor throughout this thesis and being so involved with your students. I would also like to thank my company supervisor Thomas Biesaart for all the standups, rubber ducking and sharing my enthusiasm throughout this thesis. I would like to thank Infi for providing me the opportunity to do this research (and doing it full time). And finally, my deepest thanks go out to my dearest, Maja Bloemraad. Thank you for standing by me and believing in me throughout this whole master study.

Oh and yes David it is finished now.

Bibliography

[1] ISO Central Secretary, "Software and systems engineering — software testing — part 1: Concepts and definitions," en, International Organization for Standardization, Geneva, CH, Standard ISO/IEC/IEEE 29119-1:2013, 2013. [Online]. Available: https://www.iso.org/standard/45142.html.

[2] A. Bertolino, "Software testing research: Achievements, challenges, dreams," *FoSE 2007: Future of Software Engineering*, pp. 85–103, 2007. DOI: 10.1109/FOSE.2007.25.

[3] *Junit*, https://junit.org/junit5/, Accessed on 10-04-2021.

[4] *Testng*, https://testng.org/doc/, Accessed on 10-04-2021.

[5] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli, "On the relation of test smells to software code quality," *Proceedings - 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018*, pp. 1–12, 2018. DOI: 10.1109/ICSME.2018.00010.

[6] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. Van Deursen, "Mining software repositories to study co-evolution of production & test code," *Proceedings of the 1st International Conference on Software Testing, Verification and Validation, ICST 2008*, pp. 220–229, 2008. DOI: 10.1109/ICST.2008.47.

[7] F. Horváth, B. Vancsics, L. Vidács, Á. Beszédes, D. Tengeri, T. Gergely, and T. Gyimóthy, "Test suite evaluation using code coverage based metrics," *CEUR Workshop Proceedings*, vol. 1525, pp. 46–60, 2015, ISSN: 16130073.

[8] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, vol. 16-21-Nove, pp. 654–665, 2014. DOI: 10.1145/2635868.2635929.

[9] R. J. Lipton and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978, ISSN: 00189162. DOI: 10.1109/C-M.1978.218136.

[10] A. V. Pizzoleto, F. C. Ferrari, J. Offutt, L. Fernandes, and M. Ribeiro, "A systematic literature review of techniques and metrics to reduce the cost of mutation testing," *Journal of Systems and Software*, vol. 157, p. 110 388, 2019, ISSN: 01641212. DOI: 10.1016/j.jss.2019.07.100. [Online]. Available: https://doi.org/10.1016/j.jss.2019.07.100.

[11] M. Kintis, "Effective Methods to Tackle the Equivalent Mutant Problem when Testing Software with Mutation," no. June, 2016.

[12] X. Yao, M. Harman, and Y. Jia, "A study of equivalent and stubborn mutation operators using human analysis of equivalence," *Proceedings - International Conference on Software Engineering*, vol. 1, no. 1, pp. 919–930, 2014, ISSN: 02705257. DOI: 10.1145/2568225.2568265.

[13] Y. S. Ma and S. W. Kim, "Mutation testing cost reduction by clustering overlapped mutants," *Journal of Systems and Software*, vol. 115, pp. 18–30, 2016, ISSN: 01641212. DOI: 10.1016/j.jss.2016.01.007. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2016.01.007.

[14] M. Yu and Y. S. Ma, "Possibility of cost reduction by mutant clustering according to the clustering scope," *Software Testing Verification and Reliability*, vol. 29, no. 1-2, 2019, ISSN: 10991689. DOI: 10.1002/stvr.1692.

[15] C. Ji, Z. Chen, B. Xu, and Z. Zhao, "A novel method of mutation clustering based on domain analysis," *Proceedings of the 21st International Conference on Software Engineering and Knowledge Engineering, SEKE 2009*, pp. 422–425, 2009.

[16] A. Wiliński, "Toward Generalization of Mutant Clustering Results in Mutation Testing," *Advances in Intelligent Systems and Computing*, vol. 342, pp. v–vi, 2015, ISSN: 21945357. DOI: 10.1007/978-3-319-15147-2.

[17] C. Wei, X. Yao, D. Gong, and H. Liu, "Spectral clustering based mutant reduction for mutation testing," *Information and Software Technology*, vol. 132, no. November 2020, p. 106 502, 2021, ISSN: 09505849. DOI: 10.1016/j.infsof.2020.106502. [Online]. Available: https://doi.org/10.1016/j.infsof.2020.106502.

[18] *Tiobe index*, https://www.tiobe.com/tiobe-index/, Accessed on 11-04-2021.

[19] M. Z. Rodriguez, C. H. Comin, D. Casanova, O. M. Bruno, D. R. Amancio, L. d. F. Costa, and F. A. Rodrigues, *Clustering algorithms: A comparative approach*, 1. 2019, vol. 14, pp. 1–34, ISBN: 1111111111. DOI: 10.1371/journal.pone.0210236.

[20] X. Wu, V. Kumar, Q. J. Ross, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, P. S. Yu, Z. H. Zhou, M. Steinbach, D. J. Hand, and D. Steinberg, *Top 10 algorithms in data mining*, 1. 2008, vol. 14, pp. 1–37, ISBN: 1011500701. DOI: 10.1007/s10115-007-0114-2.

[21] J. A. Hartigan and M. A. Wong, "Algorithm AS 136: A K-Means Clustering Algorithm," *Applied Statistics*, vol. 28, no. 1, p. 100, 1979, ISSN: 00359254. DOI: 10.2307/2346830. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.2307/2346830.

[22] W. S. Sarle, L. Kaufman, and P. J. Rousseeuw, "Finding Groups in Data: An Introduction to Cluster Analysis.," *Journal of the American Statistical Association*, vol. 86, no. 415, p. 830, Sep. 1991, ISSN: 01621459. DOI: 10.2307/2290430. [Online]. Available: https://www.jstor.org/stable/2290430?origin=crossref.

[23] C. Fraley, "How Many Clusters? Which Clustering Method? Answers Via Model-Based Cluster Analysis," *The Computer Journal*, vol. 41, no. 8, pp. 578–588, Aug. 1998, ISSN: 0010-4620. DOI: 10.1093/comjnl/41.8.578. [Online]. Available: https://academic.oup.com/comjnl/article-lookup/doi/10.1093/comjnl/41.8.578.

[24] T. Fv, "The Expectatio Maximization Algorithm," pp. 47–60, 1996.

[25] M. Filippone, F. Camastra, F. Masulli, and S. Rovetta, "A survey of kernel and spectral methods for clustering," *Pattern Recognition*, vol. 41, no. 1, pp. 176–190, 2008, ISSN: 00313203. DOI: 10.1016/j.patcog.2007.05.018.

[26] B. M and V. R. P, "SUBSPACE CLUSTERING ON HIGH DIMENSIONAL DATA," *i-manager's Journal on Cloud Computing*, vol. 3, no. 3, p. 18, 2016, ISSN: 2349-6835. DOI: 10.26634/jcc.3.3.8297. [Online]. Available: http://www.imanagerpublications.com/article/8297.

[27] M. Daszykowski and B. Walczak, "Density-Based Clustering Methods," *Comprehensive Chemometrics*, vol. 2, pp. 635–654, 2009. DOI: 10.1016/B978-044452701-1.00067-3.

[28] M. Ankerst, M. M. Breunig, H. P. Kriegel, and J. Sander, "OPTICS: Ordering Points to Identify the Clustering Structure," *SIGMOD Record (ACM Special Interest Group on Management of Data)*, vol. 28, no. 2, pp. 49–60, 1999, ISSN: 01635808. DOI: 10.1145/304181.304187.

[29] M. W. Berry, A. Mohamed, and B. W. Yap, Eds., *Supervised and Unsupervised Learning for Data Science*. Springer International Publishing, 2020. DOI: 10.1007/978-3-030-22475-2. [Online]. Available: https://doi.org/10.1007%5C%2F978-3-030-22475-2.

[30] K. L. Du, "Clustering: A neural network approach," *Neural Networks*, vol. 23, no. 1, pp. 89–107, 2010, ISSN: 08936080. DOI: 10.1016/j.neunet.2009.08.007.

[31] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis, and N. Malevris, "Analysing and comparing the effectiveness of mutation testing tools: A manual study," *Proceedings - 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation, SCAM 2016*, pp. 147–156, 2016. DOI: 10.1109/SCAM.2016.28.

[32] A. Márki and B. Lindström, "Mutation tools for Java," *Proceedings of the ACM Symposium on Applied Computing*, vol. Part F1280, pp. 1364–1371, 2017. DOI: 10.1145/3019612.3019825.

[33] P. Ammann, M. E. Delamaro, and J. Offutt, "Establishing theoretical minimal sets of mutants," *Proceedings - IEEE 7th International Conference on Software Testing, Verification and Validation, ICST 2014*, pp. 21–30, 2014. DOI: 10.1109/ICST.2014.13.

[34] T. Laurent, M. Papadakis, M. Kintis, C. Henard, Y. L. Traon, and A. Ventresque, "Assessing and Improving the Mutation Testing Practice of PIT," *Proceedings - 10th IEEE International Conference on Software Testing, Verification and Validation, ICST 2017*, pp. 430–435, 2017. DOI: 10.1109/ICST.2017.47.

[35] *Pit homepage*, https://github.com/LaurentTho3/ExtendedPitest, Accessed on 12-04-2021.

[36] *Pit release page*, https://github.com/hcoles/pitest/releases, Accessed on 12-04-2021.

[37] *Major mutation testing*, https://mutation-testing.org/downloads/, Accessed on 12-04-2021.

[38] *Mujava*, https://cs.gmu.edu/~offutt/mujava/, Accessed on 12-04-2021.

[39] *Pit homepage*, https://pitest.org/, Accessed on 12-04-2021.

[40] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia, and L. Zhang, "Predictive Mutation Testing," *IEEE Transactions on Software Engineering*, vol. 45, no. 9, pp. 898–918, 2019, ISSN: 19393520. DOI: 10.1109/TSE.2018.2809496.

[41] S. Oonk, "Combining Mutation Testing Strategies : Trade-Offs Between Duration and Precision," no. January, 2021.

[42] *Pit bytecode mutants*, https://github.com/hcoles/pitest/issues/391, Accessed on 12-05-2021.

[43] W. H. Gomma and A. A. Fahmy, "A Survey of Text Similarity Approaches," *International Journal of Computer Applications*, vol. 68, no. 13, pp. 13–18, 2013.

[44] *Pit mutators*, http://pitest.org/quickstart/mutators/, Accessed on 12-05-2021.

[45] A. J. Offutt and R. H. Untch, "Mutation 2000: Uniting the Orthogonal," *Mutation Testing for the New Century*, pp. 34–44, 2001. DOI: 10.1007/978-1-4757-5939-6{_}7.

[46] Oracle, *The java® virtual machine specification*, version 12, Accessed on 26-05-2021. [Online]. Available: https://docs.oracle.com/javase/specs/jvms/se12/html/index.html.

[47] *Pit plugins*, http://pitest.org/quickstart/advanced/, Accessed on 12-05-2021.

[48] N. Rajalingam and K. Ranjini, "Hierarchical Clustering Algorithm- A Comparative Study," *International ...*, vol. 19, no. 3, pp. 42–46, 2011. [Online]. Available: http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Hierarchical+Clustering+Algorithm+-+A+Comparative+Study#0%5Cnhttp://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Hierarchical+Clustering+Algorithm-+A+Comparative+Study%230.

[49] E. Gothai and P. Balasubramanie, "Performance evaluation of hierarchical clustering algorithms," *Proceedings of 2010 International Conference on Communication and Computational Intelligence, INCOCCI-2010*, pp. 457–460, 2010.

[50] V. Vijaya, S. Sharma, and N. Batra, "Comparative Study of Single Linkage, Complete Linkage, and Ward Method of Agglomerative Clustering," *Proceedings of the International Conference on Machine Learning, Big Data, Cloud and Parallel Computing: Trends, Prespectives and Prospects, COMITCon 2019*, pp. 568–573, 2019. DOI: 10.1109/COMITCon.2019.8862232.

[51] M. Dahm, "Byte code engineering," in *JIT'99*, C. H. Cap, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 267–277, ISBN: 978-3-642-60247-4.

[52] *Commons text levenshtein distance*, https://commons.apache.org/proper/commons-text/apidocs/org/apache/commons/text/similarity/LevenshteinDistance.html, Accessed on 12-05-2021.

[53] K. Potdar, T. S., and C. D., "A Comparative Study of Categorical Variable Encoding Techniques for Neural Network Classifiers," *International Journal of Computer Applications*, vol. 175, no. 4, pp. 7–9, 2017. DOI: 10.5120/ijca2017915495.

[54] S. L. Chiu, *Fuzzy model identification based on cluster estimation*, 1994. DOI: 10.3233/IFS-1994-2306.

[55] S. A. Mingoti and J. O. Lima, "Comparing SOM neural network with Fuzzy c-means, K-means and traditional hierarchical clustering algorithms," *European Journal of Operational Research*, vol. 174, no. 3, pp. 1742–1759, 2006, ISSN: 03772217. DOI: 10.1016/j.ejor.2005.03.039.

[56] I. Guyon, "A scaling law for the validation-set training-set size ratio," *AT&T Bell Laboratories*, pp. 1–11, 1997. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.33.1337&rep=rep1&type=pdf.

[57] R. Basarat, *Github repo of this thesis*, 2021. [Online]. Available: https://github.com/Rbasarat/thesis-mutation-testing/.

[58] J. C. Bezdek, R. Ehrlich, and W. Full, "FCM: The fuzzy c-means clustering algorithm," *Computers and Geosciences*, vol. 10, no. 2-3, pp. 191–203, 1984, ISSN: 00983004. DOI: 10.1016/0098-3004(84)90020-7.

[59] M. L. D. Dias, *Fuzzy-c-means: An implementation of fuzzy c-means clustering algorithm.* May 2019. DOI: 10.5281/zenodo.3066222. [Online]. Available: https://git.io/fuzzy-c-means.

[60] J. R. de Almeida Neto, L. S. Souza, and A. de Ribamar Lima Ribeiro, "Comparative analysis between the k-means and fuzzy c-means algorithms to detect UDP flood DDoS attack on a SDN/NFV environment," *WEBIST 2020 - Proceedings of the 16th International Conference on Web Information Systems and Technologies*, no. Webist, pp. 105–112, 2020. DOI: 10.5220/0010176201050112.

[61] A. Kopf, V. Fortuin, V. R. Somnath, and M. Claassen, "Mixture-of-Experts Variational Autoencoder for Clustering and Generating from Similarity-Based Representations on Single Cell Data," 2019. [Online]. Available: http://arxiv.org/abs/1910.07763.

[62] M. C. Nwadiugwu, "Gene-Based Clustering Algorithms: Comparison Between Denclue, Fuzzy-C, and BIRCH," *Bioinformatics and Biology Insights*, vol. 14, pp. 1–6, 2020, ISSN: 11779322. DOI: 10.1177/1177932220909851.

[63] J. C. Bezdek, "On Cluster Validity for the Fuzzy c-Means Model," *IEEE TRANSACTIONS ON FUZZY SYSTEMS*, vol. 3, no. 3, pp. 370–379, 1995. DOI: https://doi.org/10.1109/91.413225. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/413225.

[64] K. L. Wu, "Analysis of parameter selections for fuzzy c-means," *Pattern Recognition*, vol. 45, no. 1, pp. 407–415, 2012, ISSN: 00313203. DOI: 10.1016/j.patcog.2011.07.012. [Online]. Available: http://dx.doi.org/10.1016/j.patcog.2011.07.012.

[65] Matlab, *Subclust*, 2021. [Online]. Available: https://nl.mathworks.com/help/fuzzy/subclust.html.

[66] ——, *Matlab*, 2021. [Online]. Available: https://nl.mathworks.com/products/matlab.html.

[67] L. Chen and L. Zhang, "Speeding up Mutation Testing via Regression Test Selection: An Extensive Study," *Proceedings - 2018 IEEE 11th International Conference on Software Testing, Verification and Validation, ICST 2018*, pp. 58–69, 2018. DOI: 10.1109/ICST.2018.00016.

[68] R. E. Kirk, "Practical significance: A concept whose time has come," *Educational and Psychological Measurement*, vol. 56, no. 5, pp. 746–759, 1996, ISSN: 00131644. DOI: 10.1177/0013164496056005002.

[69] F. Emmert-Streib and M. Dehmer, "Understanding Statistical Hypothesis Testing: The Logic of Statistical Inference," *Machine Learning and Knowledge Extraction*, vol. 1, no. 3, pp. 945–961, 2019. DOI: 10.3390/make1030054.

[70] S. Hussain, "Mutation Clustering," *Masters Thesis, King's College London, UK*, 2008.

Acronyms

CLI Command Line Interface. 13, 22
CSV comma-separated values. 16
FCM fuzzy c-means. 1, 20, 21, 24, 33, 35, 37, 38, 42
LVQ learning vector quantization. 8
POC proof-of-concept. 5
SOM self organizing map. 8, 9, 20, 42
SUT System Under Test. 4

Appendix A

PIT maven code

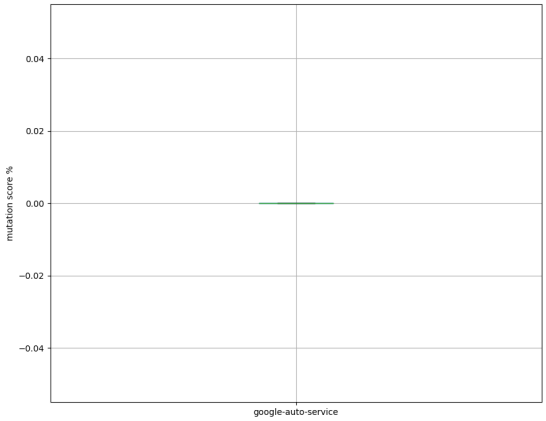
Example of code used to execute a full set of mutants with PIT.

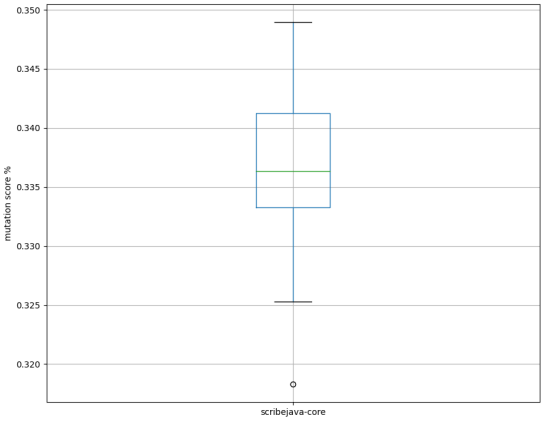
```
<plugins>
  <plugin>
    <groupId>org.pitest</groupId>
    <artifactId>pitest-maven</artifactId>
    <version>1.6.4</version>
    <dependencies>
      <dependency>
        <groupId>com.niverhawk</groupId>
        <artifactId>pitest-clustering-plugin</artifactId>
        <version>1.0-SNAPSHOT</version>
      </dependency>
    </dependencies>
    <configuration>
      <exportLineCoverage>true</exportLineCoverage>
      <mutators>
        <mutator>ALL</mutator>
      </mutators>
      <threads>6</threads>
    </configuration>
  </plugin>
```

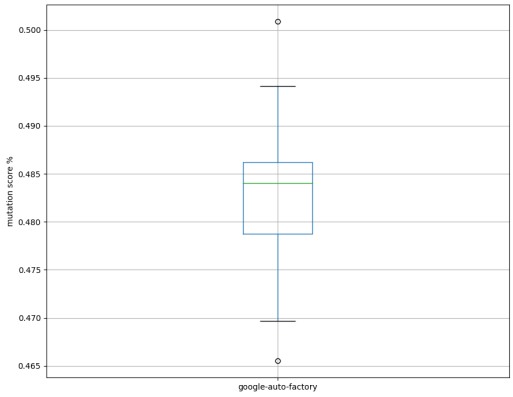
Appendix B

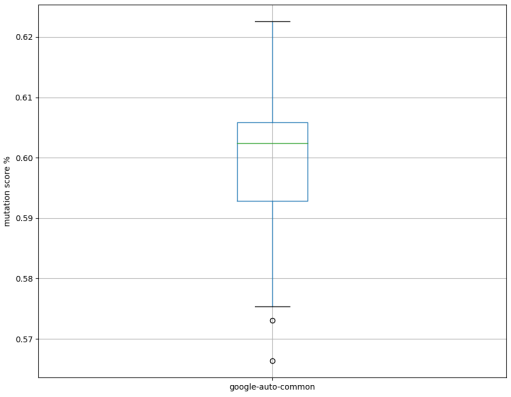
Box plots

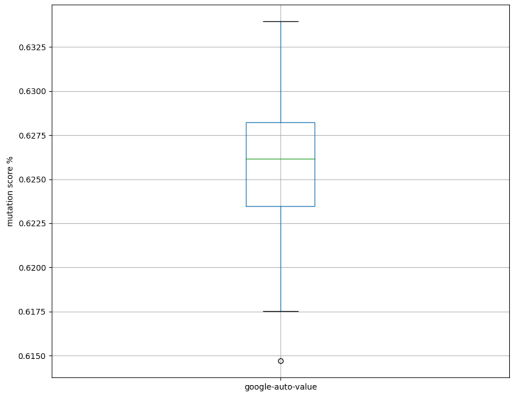
B.1 Box-plots of samples per project with all characteristics with n*0.25 reduction

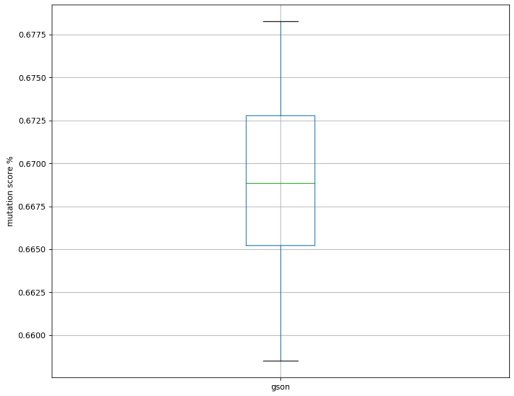


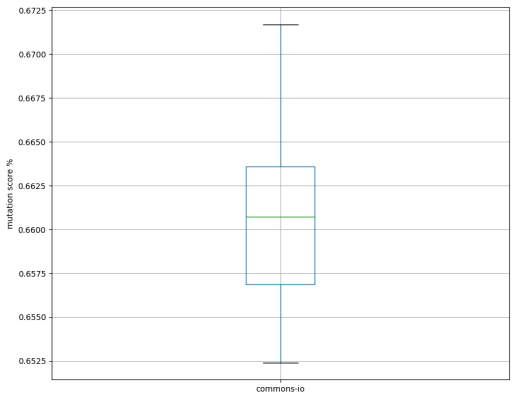


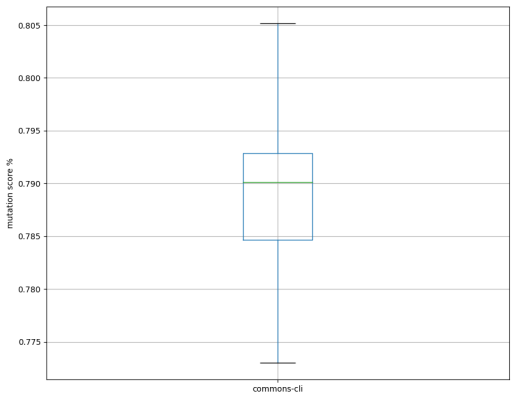


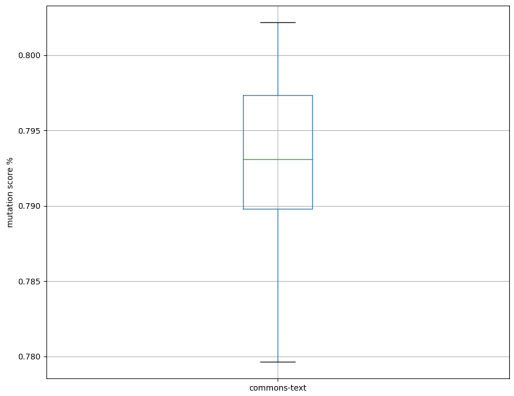


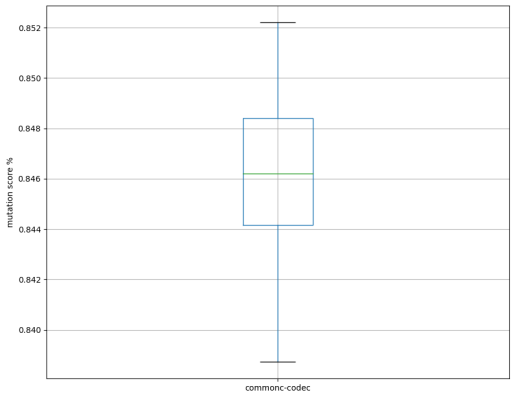

mutation score %
0.6325
0.6300
0.6275
0.6250
0.6225
0.6200
0.6175
0.6150
google-auto-value

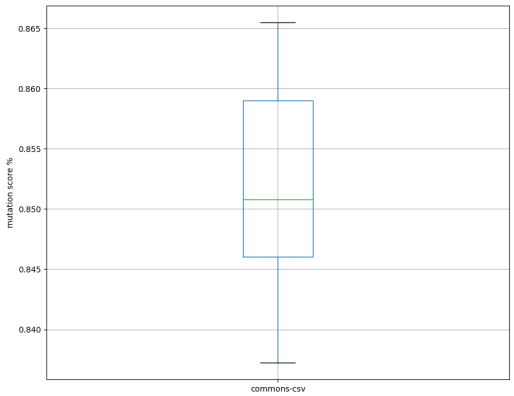

mutation score %
0.6775
0.6750
0.6725
0.6700
0.6675
0.6650
0.6625
0.6600
gson


mutation score %
0.6725
0.6700
0.6675
0.6650
0.6625
0.6600
0.6575
0.6550
0.6525
commons-io

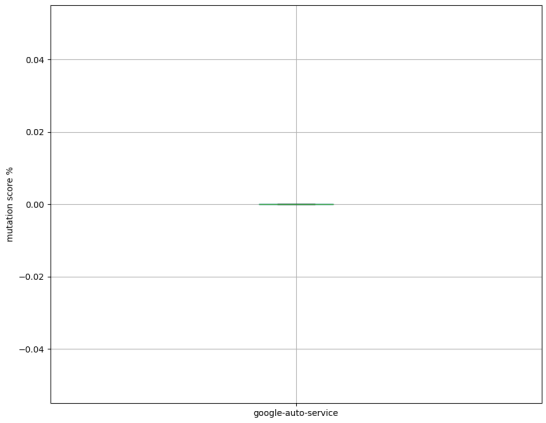

mutation score %
0.805
0.800
0.795
0.790
0.785
0.780
0.775
commons-cli

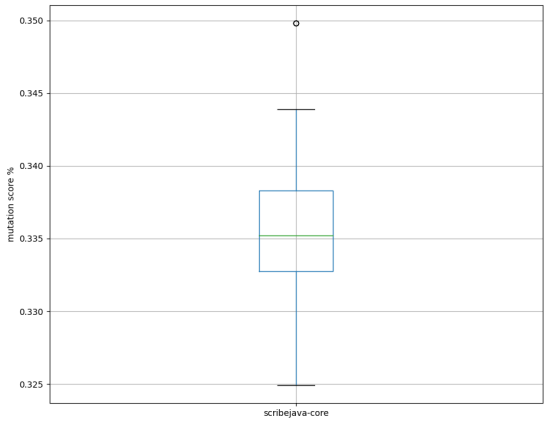

mutation score %
0.800
0.795
0.790
0.785
0.780
commons-text

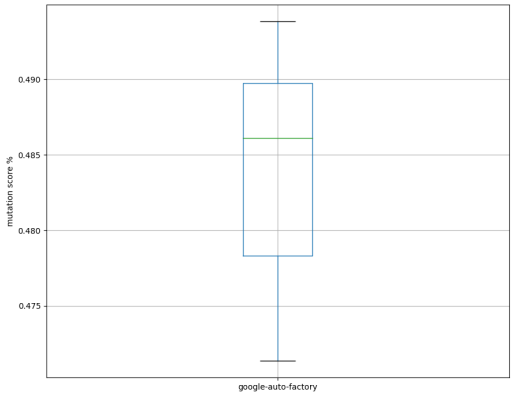

mutation score %
0.852
0.850
0.848
0.846
0.844
0.842
0.840
commonc-codec

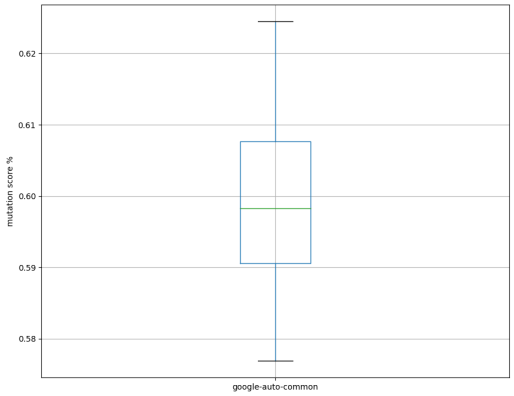

mutation score %
0.865
0.860
0.855
0.850
0.845
0.840
commons-csv

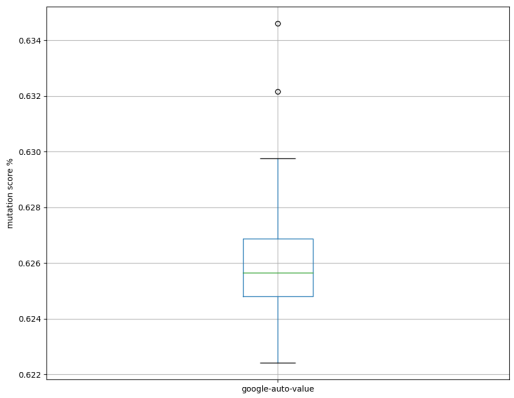
B.2 Box-plots of samples per project with all characteristics with n*0.50 reduction

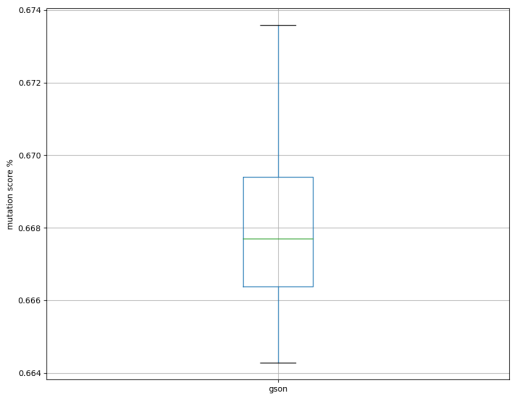


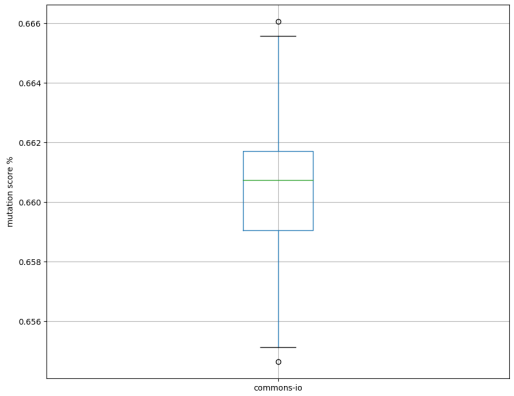


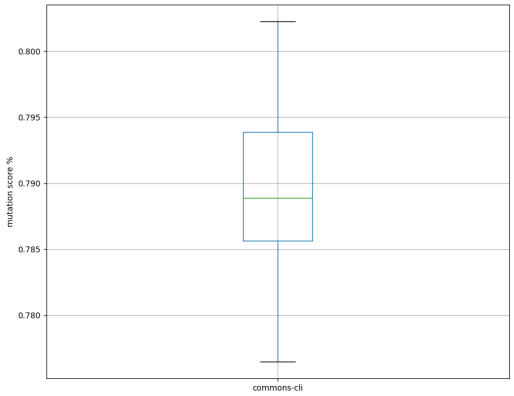


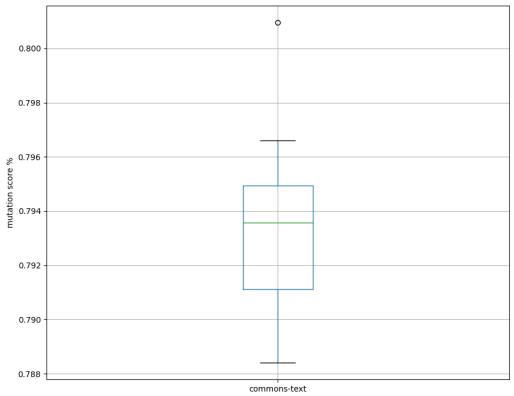


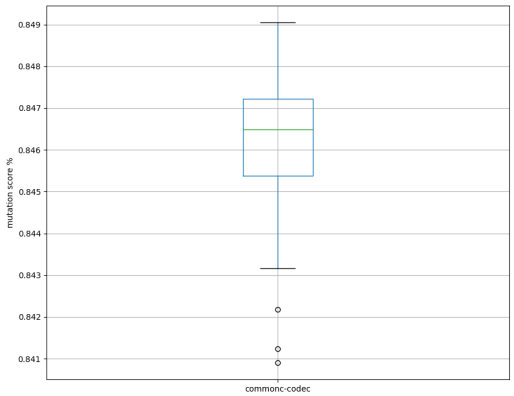


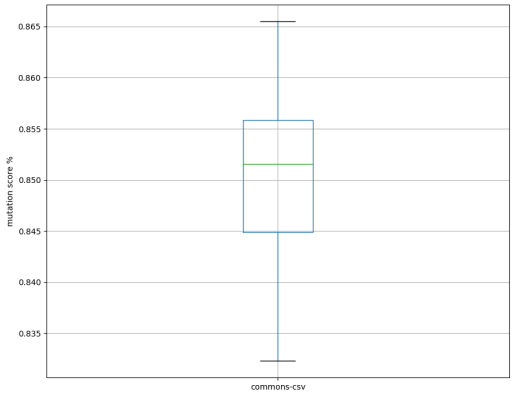



mutation score %
0.666
0.664
0.662
0.660
0.658
0.656
commons-io

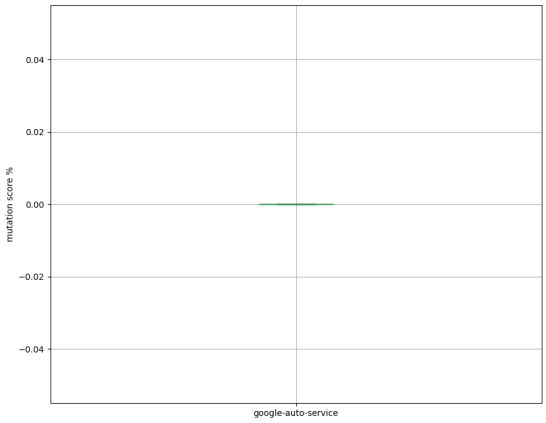

mutation score %
0.800
0.795
0.790
0.785
0.780
commons-cli

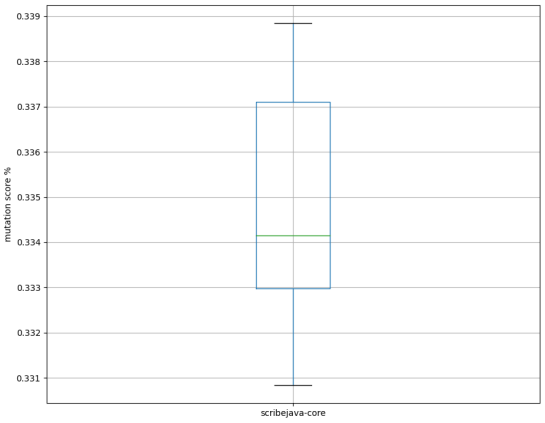

mutation score %
0.800
0.798
0.796
0.794
0.792
0.790
0.788
commons-text

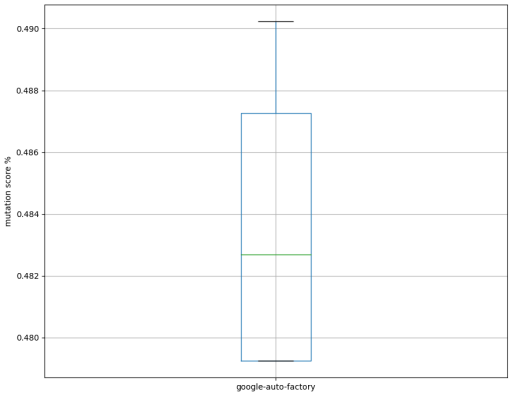

mutation score %
0.849
0.848
0.847
0.846
0.845
0.844
0.843
0.842
0.841
commonc-codec

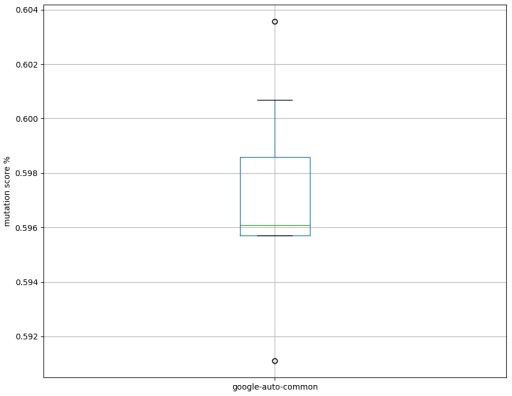

mutation score %
0.865
0.860
0.855
0.850
0.845
0.840
0.835
commons-csv

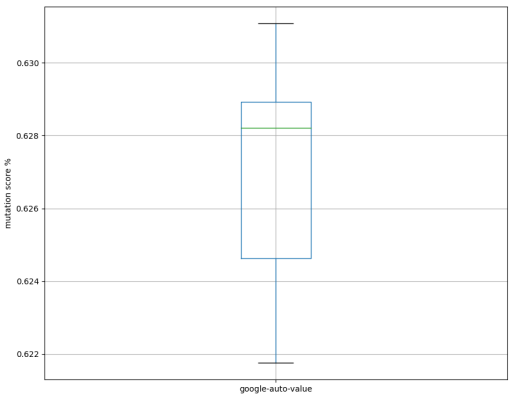
B.3 Box-plots of samples per project with all characteristics with n*0.75 reduction

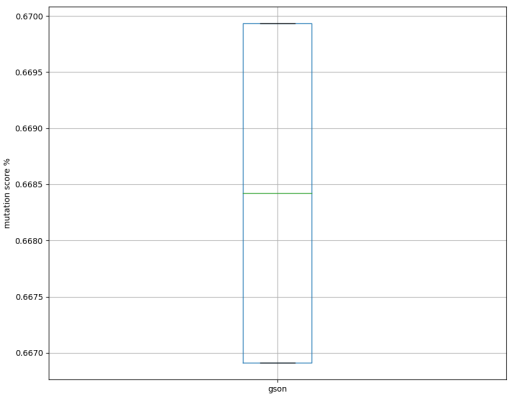


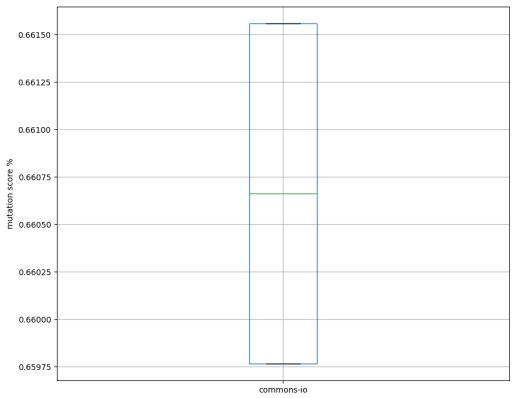


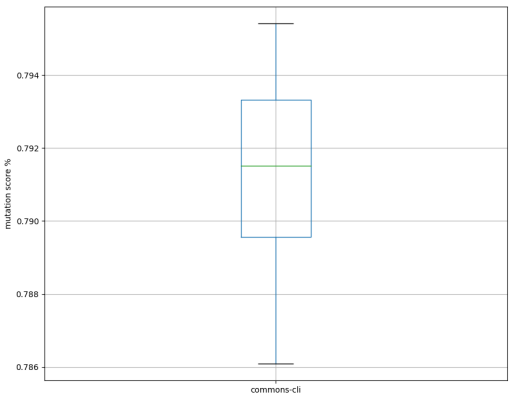


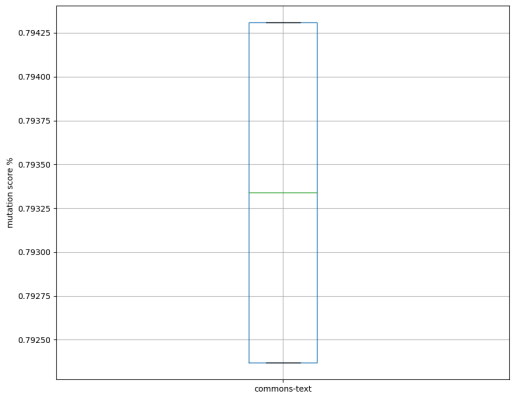


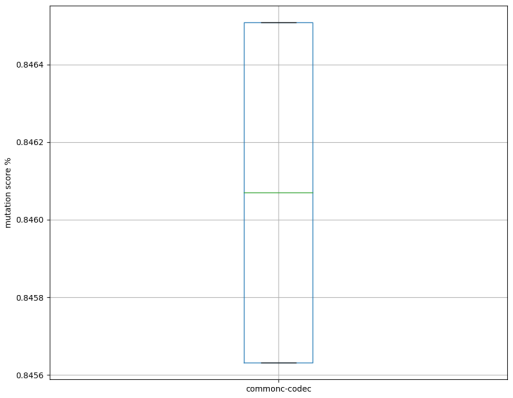


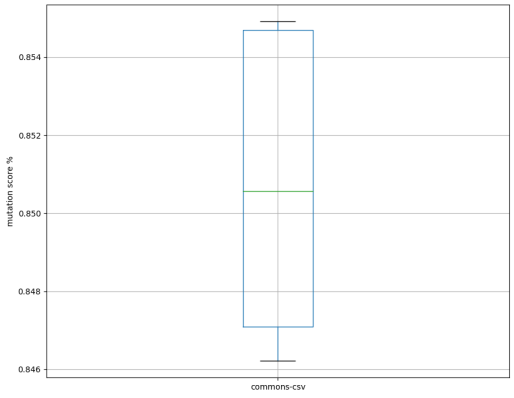



mutation score %
0.66150
0.66125
0.66100
0.66075
0.66050
0.66025
0.66000
0.65975
commons-io

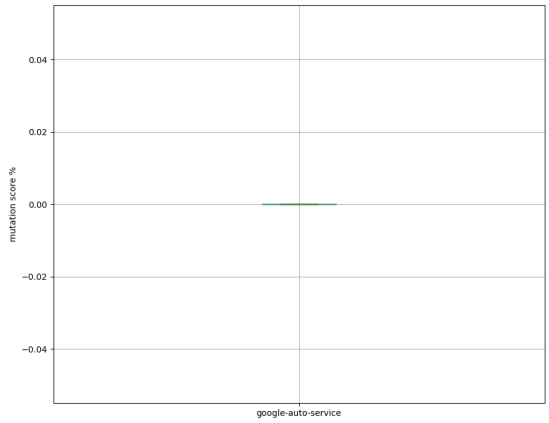

mutation score %
0.794
0.792
0.790
0.788
0.786
commons-cli

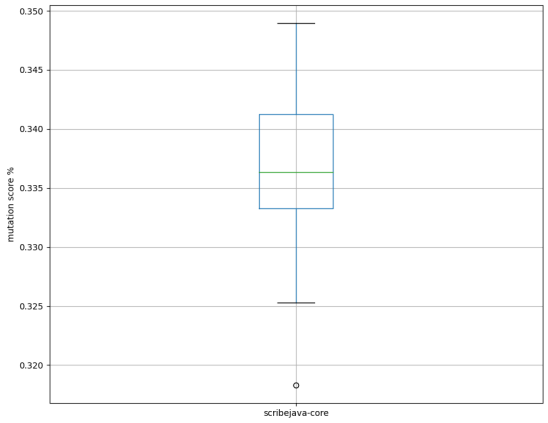

mutation score %
0.79425
0.79400
0.79375
0.79350
0.79325
0.79300
0.79275
0.79250
commons-text

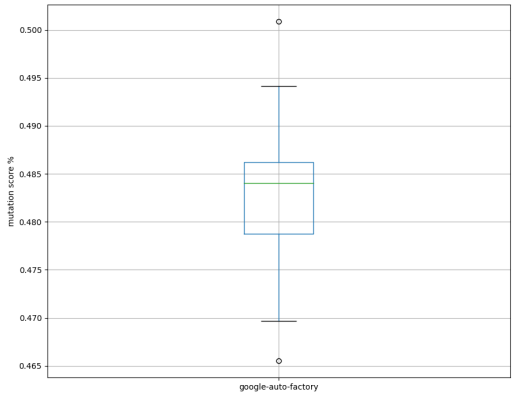

mutation score %
0.8464
0.8462
0.8460
0.8458
0.8456
commonc-codec

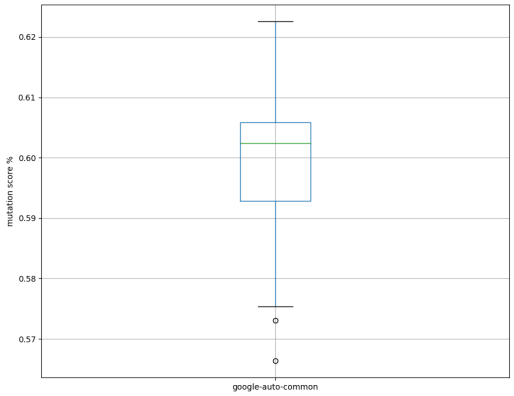

mutation score %
0.854
0.852
0.850
0.848
0.846
commons-csv

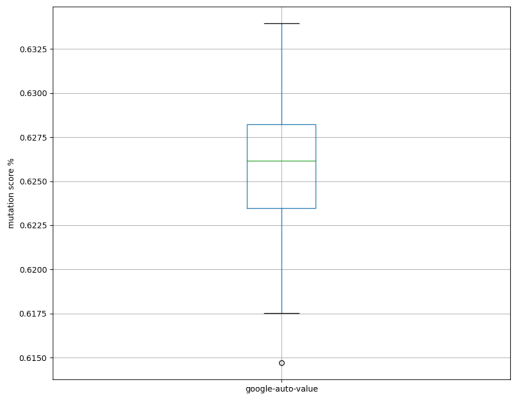
B.4 Box-plots of samples per project without Levenshtein distance characteristic with n*0.25 reduction

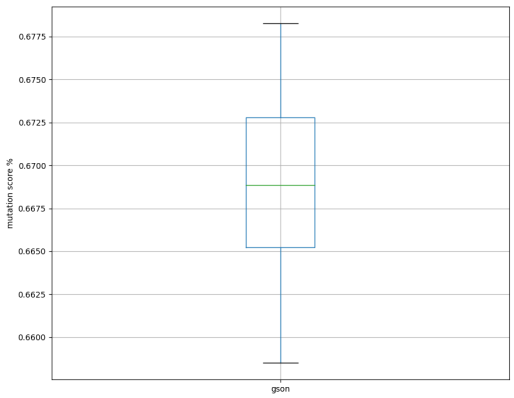


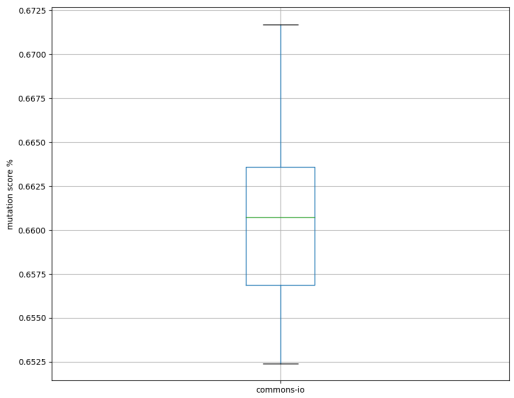


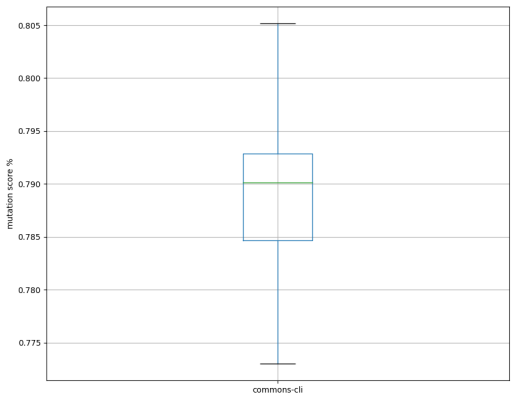


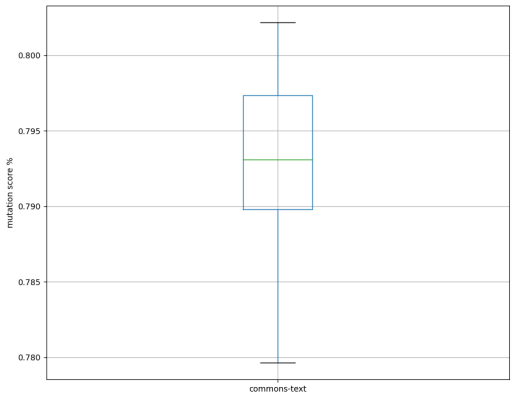


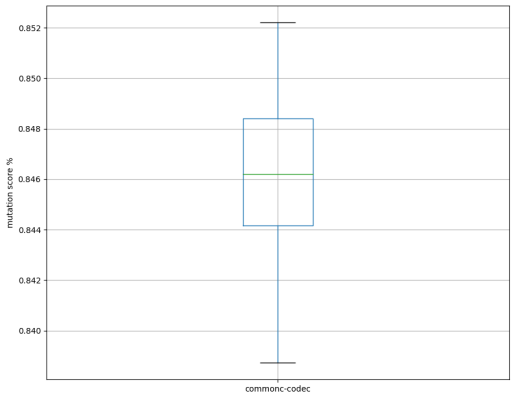


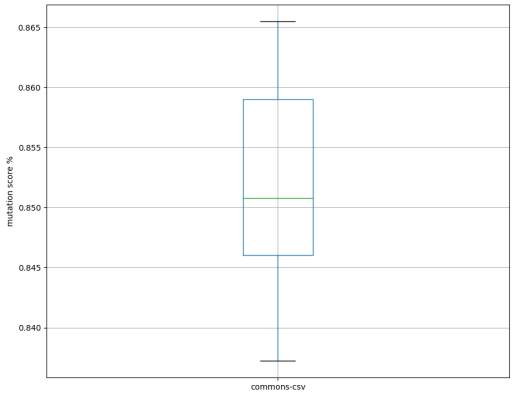



0.6725
0.6700
0.6675
0.6650
0.6625
0.6600
0.6575
0.6550
0.6525
mutation score %
commons-io

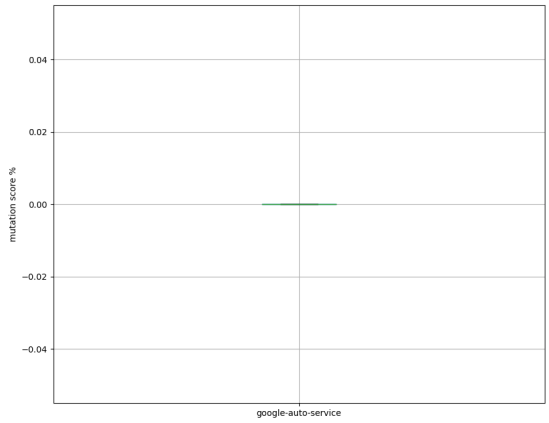

0.805
0.800
0.795
0.790
0.785
0.780
0.775
mutation score %
commons-cli

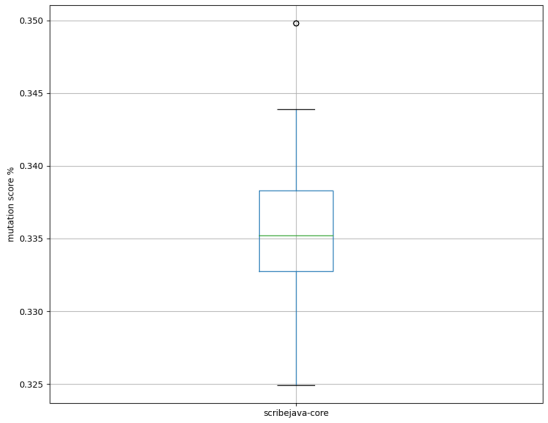

0.800
0.795
0.790
0.785
0.780
mutation score %
commons-text

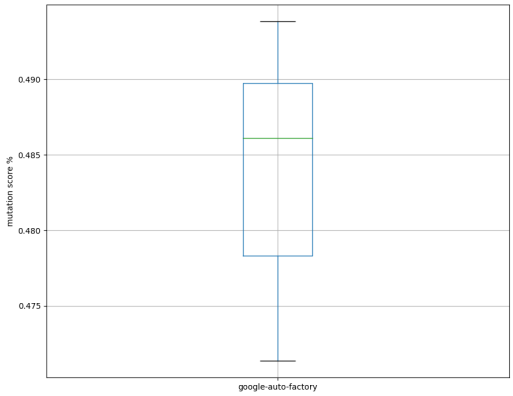

0.852
0.850
0.848
0.846
0.844
0.842
0.840
mutation score %
commonc-codec

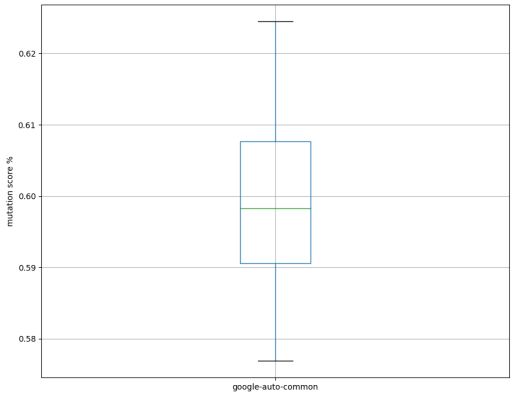

0.865
0.860
0.855
0.850
0.845
0.840
mutation score %
commons-csv

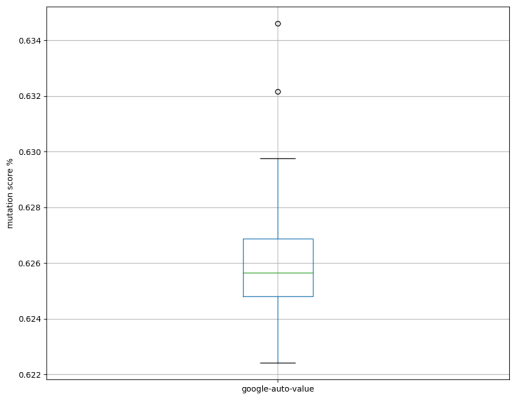
B.5 Box-plots of samples per project without Levenshtein distance characteristic with n*0.50 reduction

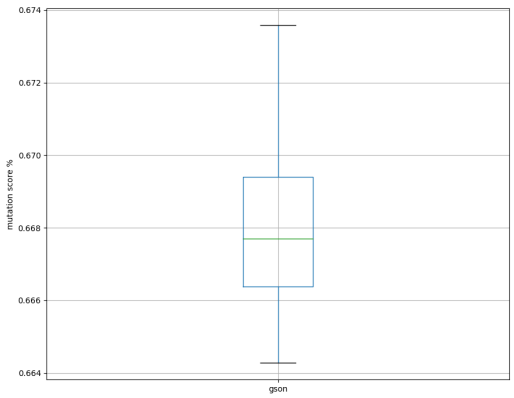




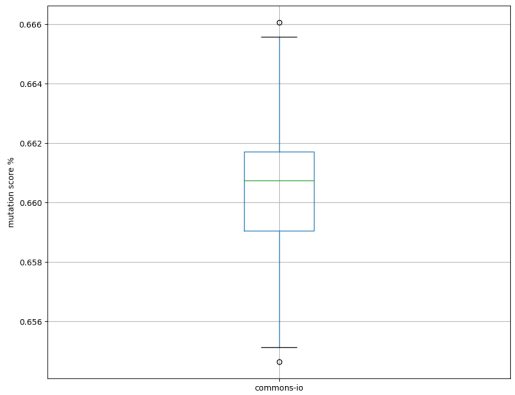


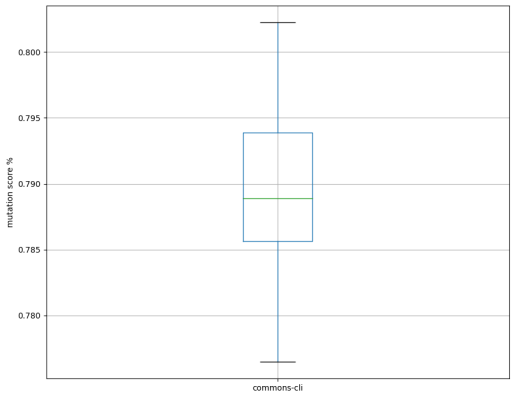


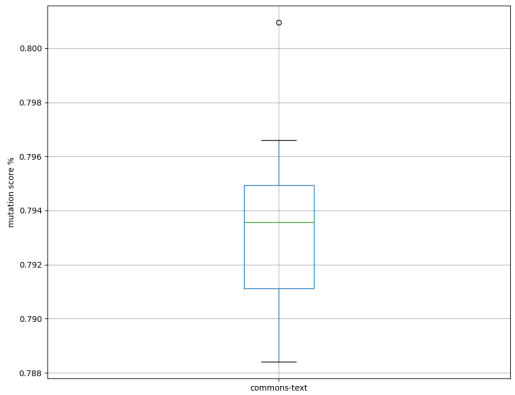


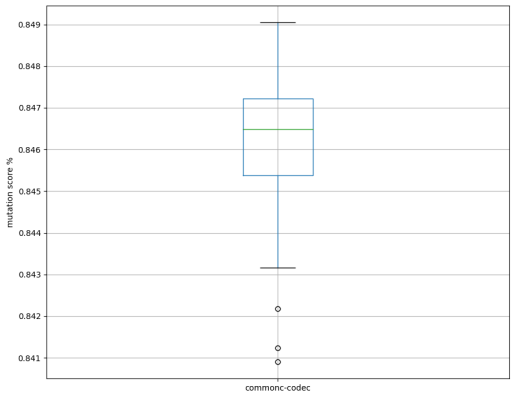


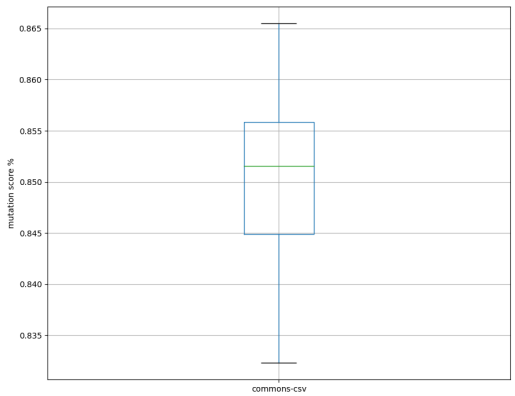
B.6 Box-plots of samples per project without Levenshtein distance characteristic with n*0.75 reduction

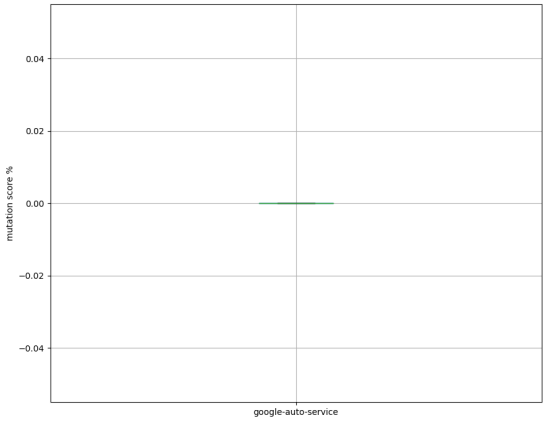

mutation score %
0.666
0.664
0.662
0.660
0.658
0.656
commons-io

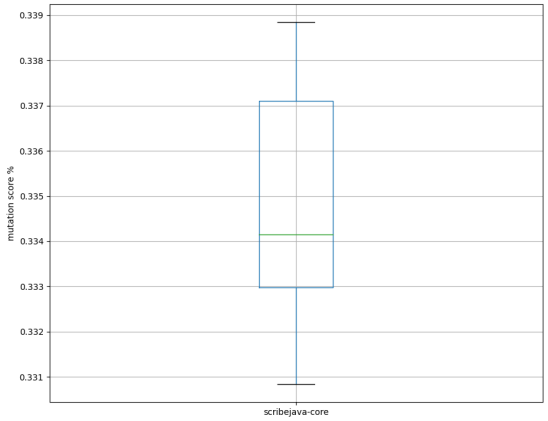

mutation score %
0.800
0.795
0.790
0.785
0.780
commons-cli

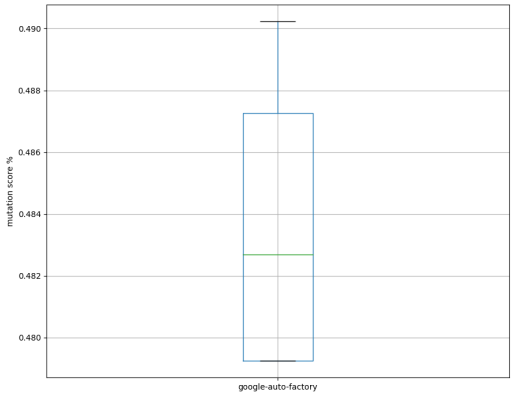

mutation score %
0.800
0.798
0.796
0.794
0.792
0.790
0.788
commons-text

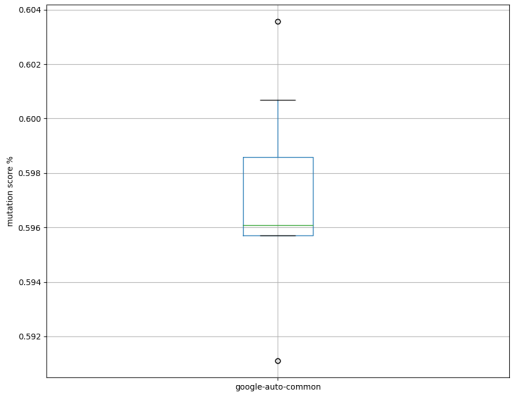

mutation score %
0.849
0.848
0.847
0.846
0.845
0.844
0.843
0.842
0.841
commonc-codec

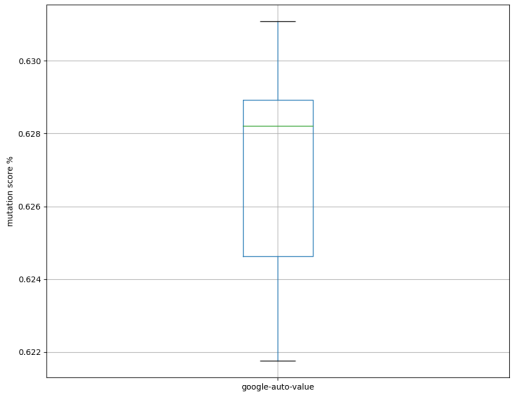

mutation score %
0.865
0.860
0.855
0.850
0.845
0.840
0.835
commons-csv

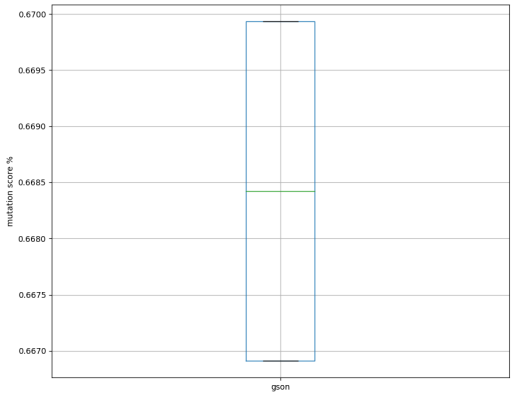

mutation score %
0.04
0.02
0.00
−0.02
−0.04
google-auto-service

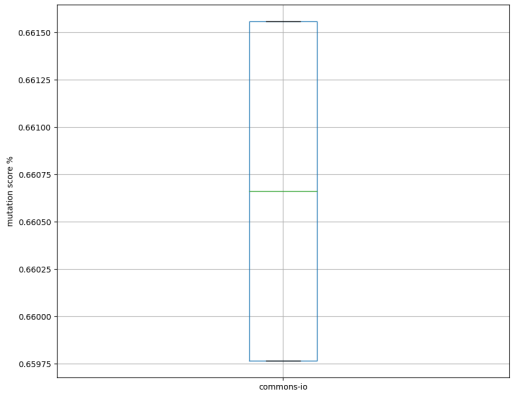

mutation score %
0.339
0.338
0.337
0.336
0.335
0.334
0.333
0.332
0.331
scribejava-core

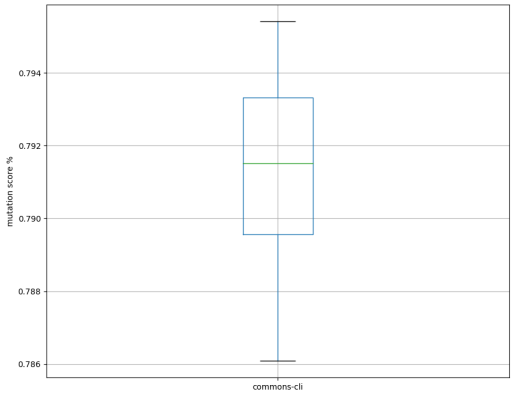

mutation score %
0.490
0.488
0.486
0.484
0.482
0.480
google-auto-factory

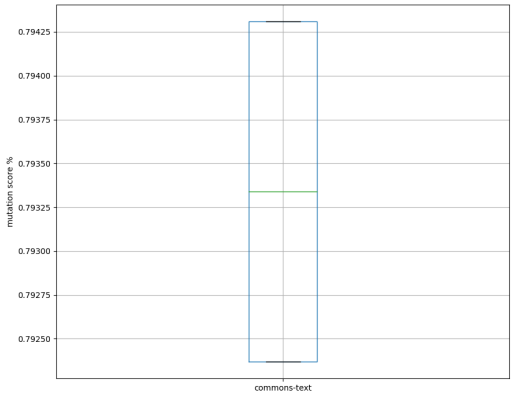

mutation score %
0.604
0.602
0.600
0.598
0.596
0.594
0.592
google-auto-common

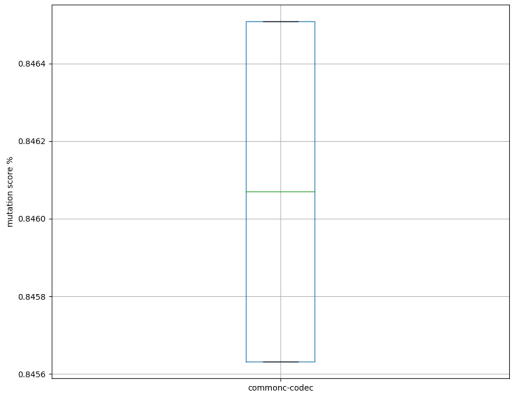

mutation score %
0.630
0.628
0.626
0.624
0.622
google-auto-value

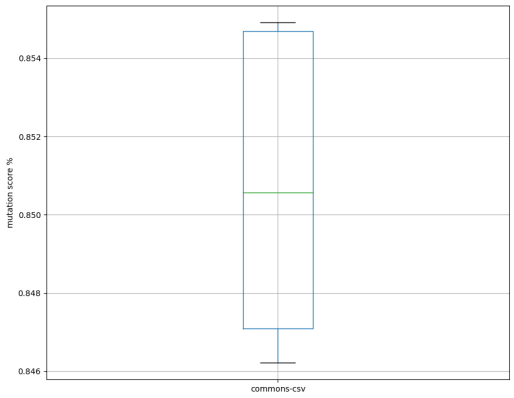

mutation score %
0.6700
0.6695
0.6690
0.6685
0.6680
0.6675
0.6670
gson


mutation score %
0.66150
0.66125
0.66100
0.66075
0.66050
0.66025
0.66000
0.65975
commons-io


mutation score %
0.794
0.792
0.790
0.788
0.786
commons-cli


mutation score %
0.79425
0.79400
0.79375
0.79350
0.79325
0.79300
0.79275
0.79250
commons-text


mutation score %
0.8464
0.8462
0.8460
0.8458
0.8456
commonc-codec


0.854
0.852
0.850
0.848
0.846
mutation score %
commons-csv