Automated Software Improvement: A Machine Learning Solution

Luke Skinner

2727141

Dr Alexander Brownlee

April 2022

Dissertation submitted in partial fulfilment for the degree of
Bachelor of Science with Honours *Software Development with Cyber Security*

Computing Science and Mathematics

University of Stirling

Abstract

Genetic Improvement (GI) of Software uses randomised search methods like genetic algorithms to apply edits to existing software. The goal of Genetic Improvement is to improve software in some way (fix bugs or improve runtime speed) and promising results have already been shown. The efficiency of Genetic Improvement could be greatly improved by targeting edits to regions of code where they are most likely to be successful. The current gap in Genetic Improvement arises from the fact that the best regions of source code are difficult to predict and thus targeting edits is a challenge.

This project attempted to use Machine Learning to address this gap and identify critical features that are suitable for predicting the performance of Genetic Improvement edits. In particular, attempts were made to identify a connection with software metrics which measure the quality or complexity of source code.

Several Machine Learning Models were compared and evaluated to determine the best model to predict regions of source code to target edits. These were trained and tested using a dataset which was comprised of results of edits made using Gin (a genetic improvement tool) over 8 open-source Java Projects and corresponding Software Metrics. This followed the CRISP-DM (**Cross-Industry Standard Process for Data Mining**) framework and investigated the relationship between existing Software Metrics and constructed Software Metrics using an iterative process.

This project managed to perform a comprehensive investigation of using Machine Learning to predict the most GI-editable regions of source code, to help guide Genetic Improvement of Software. This was one of the first projects to use JavaParser to create custom metrics to be used as features to input to the model and highlighted that there may be potential for creating more advanced metrics. 14 features were constructed and investigated using this method. There were connections found in the dataset over multiple iterations which could be investigated further. However, this failed to meet the objective of finding a reliable connection between Software Metrics and editable regions, and the models which were produced were not accurate enough to be used to predict the percentage of edits that compiled or passed successfully. These models were fitted with respected to 4 targets and the best performing had accuracy results between 3% and 8%.

Attestation

I understand the nature of plagiarism, and I am aware of the University’s academic integrity policy.

The dataset which was used and adapted, was provided by Dr Alexander Brownlee as the basis for this project.

Luke J Skinner **20/03/2022**

Acknowledgements

Firstly, I would like to give a massive thank you to my Supervisor Dr Alexander Brownlee for his incredible amount of support throughout my project and helping me gain a solid foundation in a research area that at the beginning of the project I was completely new to. Additionally, I'd like to thank my Second Marker Savi Maharaj for her feedback during the dissertation outline. I would also like to acknowledge the developers of Checkstyle and JavaParser, as these two tools were critical in carrying out this project. As well as the developers of Gin, the Genetic Improvement tool which was responsible for generating a lot of the data used in the dataset. Lastly, I would like to thank my family and friends, as well as my partner for their continued moral support throughout the project.

Table of Contents

List of Figures

List of Tables

1 Introduction

Software Engineering has advanced a great deal in recent years, and the landscape of development has evolved to include a variety of tools, frameworks, and languages. However, one thing that has not changed is the fact that developing software is a complex and laborious process, even with a team of skilled developers. There is an emphasis on both correctly identifying and fixing bugs as well as ensuring runtime is as fast as possible. There have been strides made to use Genetic Programming and Evolutionary Algorithms to tackle some of these core constraints, in a research topic known as Genetic Improvement of Software (GI). Bug fixing and optimisation can be represented as search problems, which Genetic Programming has been proven to be effective at. Additionally, Machine Learning has been used to address many real word problems from facial recognition to self-driving cars and has the potential to guide GI to make its application to such software engineering problems more efficient.

The aim of this project is to utilise Machine Learning to attempt to develop a model which will help guide this automation of software improvement to discover regions of code which are best fitted for improvement. This project will also investigate the connection between software metrics and regions of source code which are most *amenable* to GI edits. If accurate models can be produced, then this will be tested by using those results in a targeted way with the aim of reducing runtime.

1.1 Background and Context

In the attempt to develop reliable software solutions, developers can spend as much as 50% of their time testing or fixing bugs in source code [1]. This, combined with making sure that the software is optimised and performs efficiently adds a lot of overhead to any Software Development Lifecycle. This ultimately results in an increased cost [1]. Automation of tasks has helped developers in increasing their productivity, with many development tools coming with a range of features to help speed up crucial development time.

If we could target common bugs or lines of code which result in poor performance and automate the process of solving those problems this would ultimately aide in making development easier. There are many different techniques which have been researched to address this problem, with varying degrees of success. However, the application of GI of Software [2] which uses Evolutionary Algorithms to solve this problem is still being researched and has shown promising results, such as automatically fixing bugs [9] or a 70-fold speedup of a real software system [3].

GI is random in nature, and little is known about the kinds of source code it works best on. We could use the predictive power of Machine Learning to predict which regions GI works best on. This will require identifying suitable features of the source code that can be passed to the models. An easy way to obtain these features is through code quality and complexity metrics.

A solution or even a better understanding would be beneficial to the entire Software Engineering industry, as everyone benefits from the ability to make software more robust, especially if it is front facing to users. This is an interesting problem as there are many conclusions that could potentially be drawn and many applications for the results which could form the basis of further research in the future. It is important that as the complexity of technology and software increases that we address headaches that we as humans have to contend with and a solution to this problem would be a beneficial step forward.

1.2 Scope and Objectives

This project will have the following objectives:

1. Identify features that are suitable for predicting the performance of GI Improvement edits and therefore determine the regions of code most likely to offer an improvement when edited. (Which we will refer to as simply *amenable*; more fully defined in section 3.2) Particularly, attempt to identity a connection with Software Metrics.
2. Compare, evaluate, and determine the best Machine Learning model for predicting regions of code to edit using identified features.
3. Train and test the model from (2) using data generated from Gin, incorporating a range of open-source programming projects to increase generality.
4. Integrate the model from (2) within the GI toolkit Gin, so that Gin's edits can be guided to the most *amenable* regions of source code. Test experimentally to determine whether this approach improves Gin's efficiency.

Scope: Although this project will focus on determining the most *amenable* regions to edit, we will also use runtime exclusively as a target metric to validate the findings. There are other potential targets for optimisation by GI such as memory and energy reduction, which will be discussed in the next section. It is very difficult to obtain accurate, consistent measurements of memory and energy for running software, so the results with these targets may be inconclusive and therefore will not be covered here. However, identifying features to predict amenability to memory or energy improvement could be useful, so it's possible that future work could build on this project and investigate these targets further. Additionally, the functionality and results of this project will still be useful in progressing this relatively new research area.

These objectives remained mostly the same as described in the dissertation outline, however objective 1 was expanded to offer more of an explanation of the goal and point to the relevant description of *amenable* regions. Objectives 3 and 4 were rewritten to be more specific and include cross referencing.

1.3 Achievements

In this project, several features were successfully investigated with respect to predicting amenity for four targets which are based on the success of potential edits. The conclusions found by visualising, performing detailed analysis of the existing input metrics, and investigating the effect of creating custom input features and attempting to build predictive models using these features has highlighted many useful areas for future research. Although this project did not manage to produce accurate models that were capable of accurately predicting editable regions, it was able to reaffirm that this kind of prediction is inherently difficult but is still worth investigating further.

This project was one of the first to use the tool JavaParser to compute custom metrics from source code and use them as input to machine learning models of this kind.

During this project, from a personal standpoint I started with no knowledge of Genetic Improvement of Software and through this I have learned a lot about the research area. I have found that I have developed an even greater interest in the development of this area and will continue to learn more outside of this project. Additionally, I also had the opportunity to apply Machine Learning techniques to a real-world problem for the first time, in which I learned a lot about this as well.

1.4 Overview of Dissertation

The dissertation will contain the following sections and content:

Chapter 2 – State of the Art

a. This will critically evaluate the current State of the Art for the field of Genetic Improvement of Software. It will address what has been achieved so far and how this project seeks to compliment or improve on those achievements.

b. This will also discuss existing attempts to find *amenable* regions, as that is the topic of interest and the problem that this project will attempt to find a solution to or provide more details for future research.

Chapter 3-5 – Technical Chapters

a. These chapters will contain the technical content for this project.

b. Chapter 3 will give an overview of the Problem Description and Analysis, which will include explaining the problem that is being solved, as well as providing context and analysis for the requirements of the project.

c. Chapter 4 will discuss the feature construction that was performed in this project using JavaParser in more detail, as well as briefly explaining how the library functions.

d. Chapter 5 will describe the design and implementation that was carried out in the project. This will be a high-level description which is modelled after the stages in the CRISP-DM Framework. (Described in 3.4) This will also contain an evaluation of the product.

Chapter 6 – Conclusion and evaluation

a. This chapter will contain a summary of the achievements for this project for the readers convenience. These achievements will then be critically evaluated against the Scope and Objectives that was set out in the beginning of the dissertation project.

b. It will address the limitations of this project, and how those limitations may be improved on in the future. This will also attempt to guide others who may take on this project for their dissertation in the future, and how best they can utilise the results found from this project to help improve their own results.

References

a. All work which was either reviewed in the State of Art or used in this project is appropriately referenced and the credit goes to those authors.

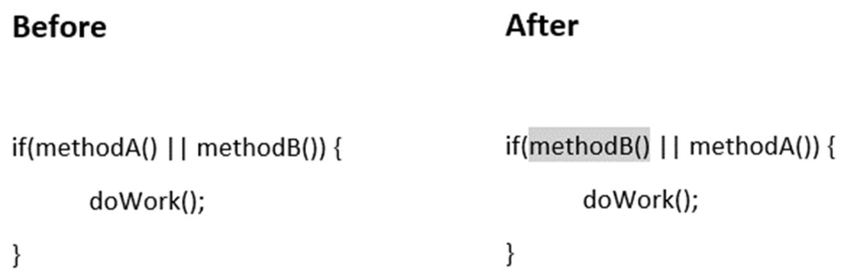
2 State-of-The-Art

This section covers the relevant State of The Art for Genetic Improvement. This will be split into three subsections.

The first subsection will critically review and discuss Genetic Improvement of Software today, and the relevant research that has been accomplished by others in the field. The second section will expand on this further by discussing the existing methods to determine *amenable* source code regions, which directly relates to the aims of this project. Finally, a conclusion will be drawn on the current state of the art and where this project aims to fill in the gaps.

2.1 Genetic Improvement of Software Today

Genetic Programming and Evolutionary Algorithms have been shown to yield great power in finding solutions to complex computational and mathematical problems. This has been applied to many real-world problems as well and has promising results across sectors. The potential is still relatively untapped in the context of software engineering and as we continue to make advances in both technology and software, so too do we need reliable solutions to problems. Genetic Improvement of Software despite still being a young research area has had a few steps forward in the recent years. The principle of creating variations of software by applying small, granular edits has been applied to several problem domains.



Before

```
if(methodA() || methodB()) {
        doWork();
}
```

After

```
if(methodB() || methodA()) {
        doWork();
}
```

Figure 1. Example of Genetic Edit Being Applied

Optimisation of Software has always been a struggle in the Software Engineering industry. There are always new attempts to help automate aspects of a developer's job, and thus ease the burden of writing reliable software solutions. It is not surprising that this would be one of the problems Genetic Improvement attempts to solve.

GI applies simple edits or mutations to source code, such as the swap illustrated in figure 1, which are then tested by running the program for real. As such, the optimisation targets the operational performance of the program in real-world usage.

This has shown success when applied to existing Software. It appears to scale well too, in fact the Genetic improvement of Software for Multiple Objective Exploration (GISMOE) approach found that when applied to a large existing code base, the automatically improved code was on average 70 times faster than the original and maintained functionality [3]. It will be the goal of this project to increase the efficiency of GI of Software, and it is possible that this could produce more consistent results when performing newer studies.

Code repair is another hurdle that developers must jump over to ensure that the applications created are both secure and functional, and this of course eats into valuable development time [1]. GI has shown interesting results in this domain as well. A large evaluation using GenProg – a genetic programming application that detects defects – was able to repair 55 out of 105 defects in C Programs across 8 open-source projects [9]. Although code repair is not the intended target of this project, it is possible the findings could still help in extending progress in this area as the same *amenable* regions could be used.

Another consideration is memory, this is often as important as runtime execution when it comes to what makes a software solution reliable. There are many memory-intensive applications, and this can sometimes be due to poor performing lines of code. Genetic improvement has seen reductions of up to 21% in memory consumption [10]. It may seem at first glance like these advances have only made waves in purely esoteric problems. However, GI of Software has also found that energy consumption can be reduced by up to 25% [11]. A link was established between less CPU cycles and the reduction of energy consumption. It is possible that a similar model to the one being proposed could help verify this link. However, that is well outside of the scope of this project.

The common theme that underpins all the research covered here is that GI edits are applied at random, and the goal is to make the choice of location more intelligent.

2.2 Existing Attempts to Determine Amenable Regions

The following sections will go into more detail about existing attempts to discover *amenable* regions of source code as well as covering more studies that have applied GI to problems and demonstrating why it is important to identify these regions. Please see section 3.2 for a more detailed description of *amenable*, as this is crucial for understanding the breadth of studies performed in this area.

2.1.1 Survey of Genetic Improvement Search Spaces

Petke et al. [12] performed a survey of Genetic Improvement search spaces, to attempt to determine an effective search methodology for *amenable* regions. Included was an analysis on the number of papers about Genetic Improvement. This survey analysed the search spaces in both functional and non-functional improvement. Despite the number of papers and clear indication that Genetic Improvement shows results in these areas, the survey found that the definition of *"effective"* mutations was not the same for functional and non-functional improvement. In general, the lack of uniformity between the papers appears to make it difficult to draw concrete conclusions on the best regions of source code to perform edits. If this project can predict what regions of source code are *amenable* with reasonable accuracy, then more uniform conclusions could be made, and it would be possible to review these kinds of improvement with these findings.

2.1.2 Neutral Program Variants

Harrand et al. [13] conducted research to attempt to discover which regions of source code are considered *"plastic"* and produce Neutral Program Variants which are variations of the source code which still behave in the same way with respect to the unit tests; see [14] by Schulte et al. for more information about neutral variants. These are important, as they represent versions of the source code where a genetic edit was performed and did not break the functionality. *Plastic* in this context refers to those regions of source code that are editable while maintaining functionality, according to the defined unit tests as stated above.

Shulte et al. [13] applied the three mutational operations add, delete, and replace. The add operation will mutate the original source code by adding a new statement or operator, the delete operation will remove an existing statement or operator and the replace operation will replace an existing statement or operator with a new one; this is illustrated in figure 2. They found that the delete statement operator was an effective edit which often maintained functionality.

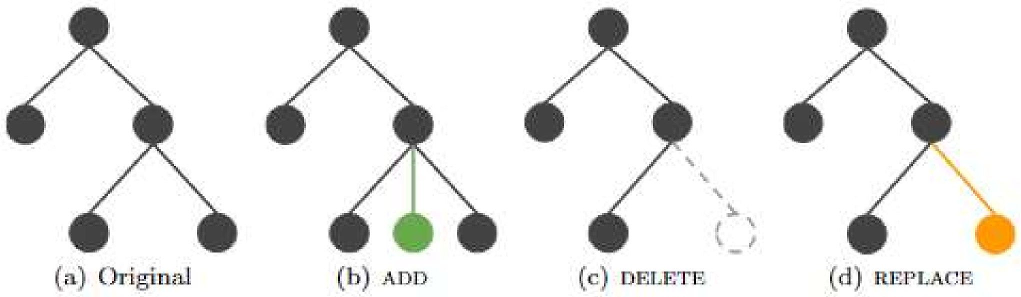


Figure 2. Program Transformation with Genetic Operators [13]

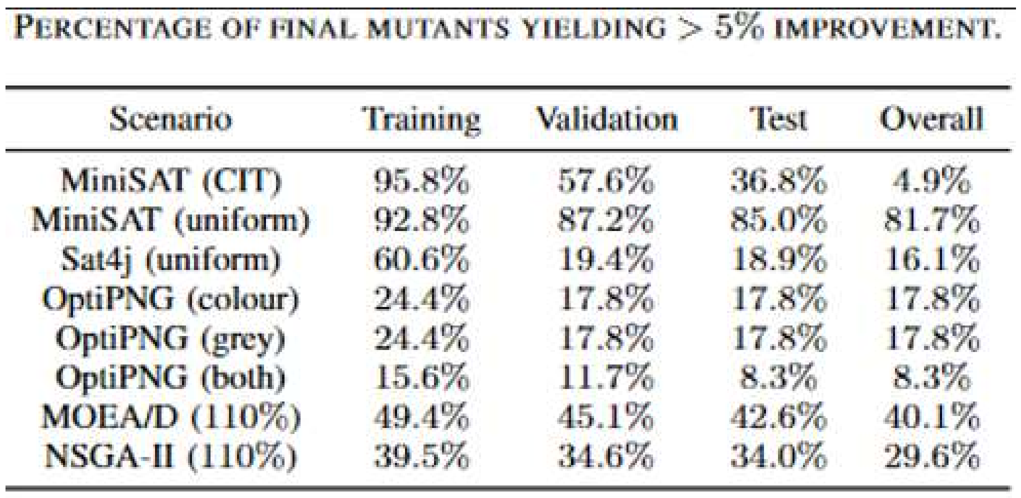
As well as this they introduced three new operators which were:

1. Add Method Invocation
2. Swap Subtype
3. Loop Flip

These showed an improvement in generating neutral variations of the software, 60%, 58% and 73% respectively. This experiment was performed across 6 open-source projects. However, it is indicated that these findings may not generalise to different kinds of applications and that this was performed on a large scale. This means there is a potential that there could have been bugs in the software or other factors that impacted the results. This project could address some of the gaps in this if accurate models are produced that generalise well to the dataset. If the results of the prediction are reliable, then the operators that were introduced could be tested using those results to determine if more neutral variants can be produced.

2.1.3 Comparison of Search Heuristics for GI of Software

Blot et al. [15] performed a comparison of search strategies which are alternative to Genetic Programming approaches. Genetic Programming has been heavily used and this paper investigates if other search strategies are more effective for GI of Software. Blot et al. [15] compared 18 search strategies with 8 different GI scenarios. They provided additional benchmarks for GI and investigated new software patches. They then compared these search strategies and assessed their effectiveness; this meaning how much each strategy performed with respect to improvement of the original source code. They found that across all the search strategies proposed as alternatives to Genetic programming methods, there was between 15% and 68% in improvement. The yield for mutants/patches that had more 5% improvement is shown in figure 3.



PERCENTAGE OF FINAL MUTANTS YIELDING > 5% IMPROVEMENT.

Scenario	Training	Validation	Test	Overall
MiniSAT (CIT)	95.8%	57.6%	36.8%	4.9%
MiniSAT (uniform)	92.8%	87.2%	85.0%	81.7%
Sat4j (uniform)	60.6%	19.4%	18.9%	16.1%
OptiPNG (colour)	24.4%	17.8%	17.8%	17.8%
OptiPNG (grey)	24.4%	17.8%	17.8%	17.8%
OptiPNG (both)	15.6%	11.7%	8.3%	8.3%
MOEA/D (110%)	49.4%	45.1%	42.6%	40.1%
NSGA-II (110%)	39.5%	34.6%	34.0%	29.6%

Figure 3. Table showing final mutants that are > 5% improvement [15]

Despite the impressive results that have been found for GI improvement in this comparison of search strategies/heuristics it is noted that there is still a large gap in finding consistent performance. This is due to the relatively young nature of the research area and that obtaining optimal performance is inherently difficult. This is an important study as it showcases that again, under the right circumstances GI of software can produce good results. It is hoped that if this project can produce accurate models, then it could be used to target edits with the best suited search method.

2.1.4 Evaluation of GI Tools for Non-functional Properties of Software

Zuo et al. [16] conducted an evaluation of GI tools with a focus on non-functional aspects of software. This consisted of a literature review of 63 papers that used GI tools for non-functional improvement of software as well as a usability study of open-source GI tools and a generalisability study of 8 different GI tools. Examples of non-functional properties are run-time, memory consumption or the size of the source code. These properties are usually important for the performance and maintainability of a software product. In this paper it was found that majority of GI tool literature considered improving run-time/execution time, as shown in figure 4.

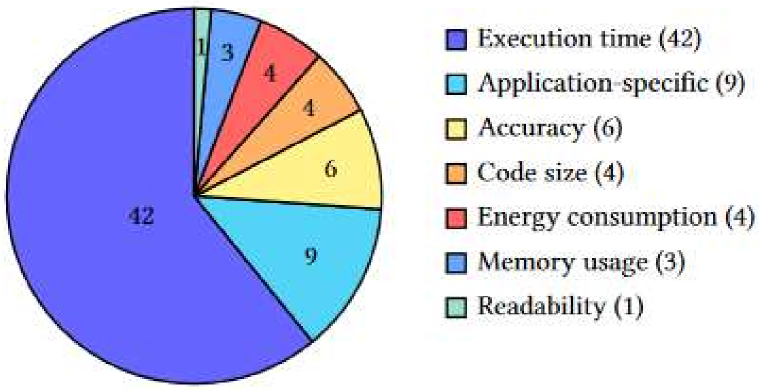


Figure 4. Distribution of non-functional aspects [16]

Execution time is an important property to improve as it affects almost every aspect from user experience to fast performance. Upon performing this survey of GI tools and their implementation, the study found that a lot of GI work does not come with implementations that can easily be reused. It also concluded, that out of the 8 GI tools which were used, only two of these could be easily ran on new software. This means that at the current time, most GI tools do not generalise well to new examples of source code. This is important as it highlights the need to discover *amenable* regions. It may be easier to create reusable implementations if it is easier to target GI edits to source code. Additionally, by focusing these on execution time, it may be possible to improve them by a larger percentage.

2.1.5 Analysis of Mutation Operator Selection Strategies for GI

Smigielska et al. [17] performed an analysis of selection strategies for mutation operators. These operators include modifying the source code via actions such insertions, deletions or replacements. This study uses a uniform strategy for selecting operators so that the search space can be explored effectively. It found this uniform strategy has the potential to improve the performance of automated program repair tools. Additionally, when analysing different mutation operators for elements of source code such as expression or declaration statements it was discovered that all operators are more successful for finding improved or maintained variants when expression statements are targeted; the deletion operator being the most successful at 50.03% respectively. The percentages of variants which maintained or improved the fitness when using the deletion operator are shown in figure 5.

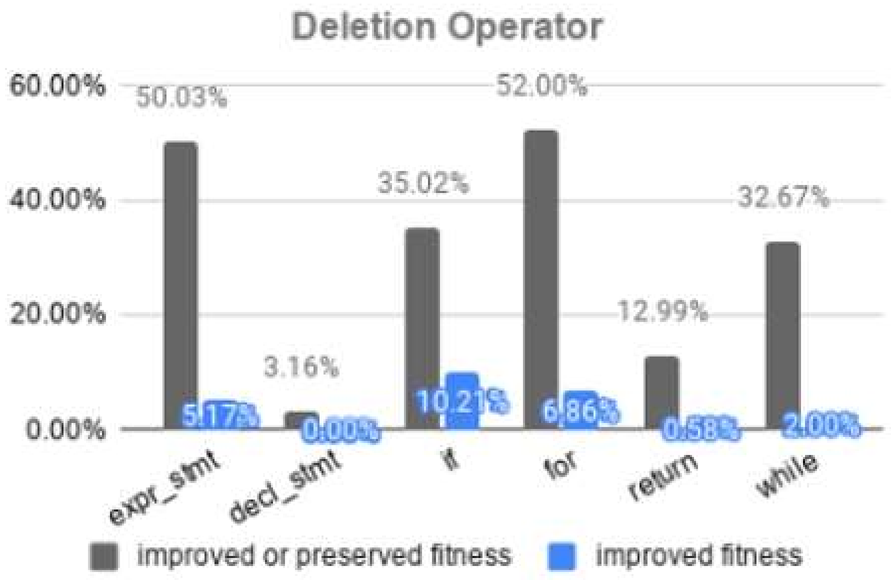


Figure 5. Percentages of Deletion Operator variants that maintained/improved fitness [17]

However, this paper states that the benchmarking used utilises small programs and does not include complex defects, which means it likely would not scale or generalise well to larger programs. Additionally, some of the results may be inflated by some of the fixes being a single incorrect operator. This study shows how GI can also be applied to functional properties and still have impressive results under the right circumstances. If more information could be gathered about the kind of source code which is most *amenable* to edits, it may be easier to create software which can generalise to both new and larger source code.

2.3 Conclusion

In conclusion, one of the major obstacles in this research area is that there is a great difficulty in determining which regions of source code are amenable, and thus the searching domain is complex and computationally expensive. The 21% reduction in memory consumption and the GISMOE approach had smaller sample sizes regarding source code analysis. [10][3] Additionally, although there were promising results from Harrand et al. [13] it is still largely unknown if the definition of *plastic* regions will hold true when applied to different applications. The survey of search spaces [12] found there is not a uniform definition for effective edits. Additionally, where there were promising results, there was a lot of human guidance involved. Therefore, if better methods of identifying *amenable* regions was discovered, this could help drive the field of Genetic Improvement to a more standardised definition of program transformation. The state of the art in Genetic Improvement appears to show good results when scaled but several studies could benefit from better predictive capability. This project will hopefully build upon the work of Harrand et al. by using Machine Learning to get us one step closer to the much-needed answer to this problem. It is possible that this will be able to verify Langdon's [3] average 70 times increase in runtime but the goal is to make the results more consistently good.

3 Problem Description and Analysis

The following sections will describe the problem and an analysis of some of the requirements for this project.

3.1 Problem Summary

This project will attempt to build predictive models to guide GI of software to a more targeted approach and using this new GI framework reduce the runtime of real-world application software . We will use Gin as the base implementation of GI. Gin is an open-source GI toolkit which creates variations of software using Evolutionary Algorithms and performs different levels of edits on source code [4][5]. This includes swapping lines, removing lines, or inserting statements. These G edits are also referred to as mutations and these follow the style of Langdon [3] and Le Goues [9], two of the most well-known GI works. In Gin, groups of edits are known as patches and can be applied at the statement level or line level.

Additionally, Gin contains a feature called RandomSampler which randomly applies edits to source code and generates a CSV file recording the results. These results contain data such as whether the edit caused the program to not compile, or in the case where it did compile whether it passed the required unit tests or not. It also contains data on execution and CPU time. The experimental data which is provided for this project is an aggregate of many random samples of edits applied to range of open-source projects and contains statistics on those edits. As the goal is to predict the most *amenable* regions that can be edited, and the data provided contains software metrics which will be used as features to the model, it is important to identify which of these are critical features.

At this point, it is important that we understand what is meant when we refer to “Amenable Regions”. The next section will describe this in more detail.

3.2 Definition of Amenable Regions

This subsection will give an overview of what is meant by *amenable* when referring to regions of source code. Note: There may be other terms used when discussing research papers that are analogous to *amenable* (such as *Plastic* or *GI-Able*), but this project will stick to this one consistent term going forward.

To help guide GI of Software, we must understand what parts of any given source code are more likely to result in a success when a genetic edit is applied to them. We can measure success of a genetic edit by looking at the proportion or percentage of the source code which compiled successfully and passed its unit tests after the edit was applied. Source code can also be characterised in many ways: including numerical measures of complexity, readability and so on. We refer to these measures as *features.* The goal is to find out which of these features predict these percentages and therefore predict the success of genetic edits.

When we refer to regions of source code that are *amenable*, we mean regions of source code which result in a successful edit. This means that we can find regions of source code to edit with less chance of breaking functionality (failing the unit tests) and explore alternative implementations of the original behaviour. This is a crucial step towards finding implementations that do the same job, but faster. Software Metrics may measure complexity or quality of source code and by examining whether these have a connection to the success of an edit, we can gauge what regions of source code are more *amenable*. For example, if a high Cyclomatic Complexity resulted in a lower success rate then we could theorise regions of source code which are less complex are more *amenable* to genetic edits.

3.3 Software Metrics

The following sections will highlight all the Software Metrics that will form features that are used as inputs to the Machine Learning models. As we want to assess whether these metrics have a connection to the success rate of genetic edits and regions that are *amenable*, it is important to understand how they are implemented.

3.3.1 Cyclomatic Complexity

Cyclomatic Complexity was introduced by Thomas J. McCabe in 1976 [6]. This is used as a quantitative metric to determine the number of individual paths through source code. The source code is represented in the form of a control-flow graph which visually articulates the paths that can be traversed in the program. The measure is formed from the elements of the graph, they are as follows:

1. G – The graph to solve for
2. E – The number of edges (the transfer arrows between nodes)
3. N – The number of nodes
4. P – The number of connected components/exit nodes

The formula can be represented as:

$$V(G) = E - N + 2P$$

McCabe [6] describes in much more detail different factors which can impact the above formula, as well as representing the graph in a more detailed form. However, to simplify consider the following basic program:

```
first = 35
IF third < second
    first = second
ELSE
    first = third
PRINT first, second and third
```

So, by analysing the control flow of the graph for this program (See Figure 2) the cyclomatic complexity of this program would be:

$$V(G) = 7 - 7 + 2 * 1 = 2$$

The lower the cyclomatic complexity the less paths to traverse through in the source code and therefore is not as computationally expensive. This metric could be important for the proposed model as potentially the more complex the program, the more likely edits are to fail.

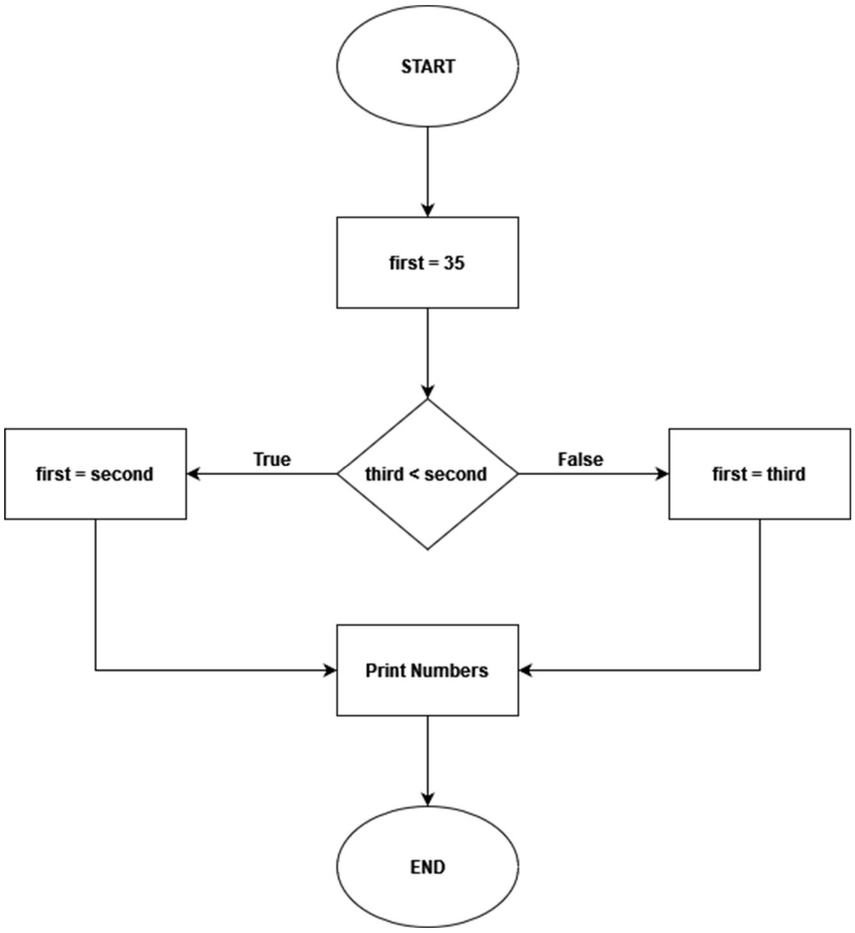


Figure 6. Control Flow Graph of Simple Program

3.3.2 NPATH

NPATH was developed as an extension to the above Cyclomatic Complexity and was introduced by Brian A. Nejmeh in 1988 [7]. This addresses the issue of multiple outcomes within source code such as if statements. Like its predecessor it measures the number of paths that can be taken through a program, with a focus on methods. To illustrate this, consider this simple method written in Java:

```
public static void foo(int first, int second)  {
        if(first > 15)  {
            System.out.println(“1”);
        }
        else {
            System.out.println(“2”);
        }
        if(first > second)  {
            System.out.println(“3”);
        }
        else {
            System.out.println(“4”);
        }
}
```

As can be seen there are two if statements, but 4 possible outcomes if you include the else blocks. NPATH does not have a simplified calculation form like Cyclomatic Complexity, but we can formulate it as:

$$N = numberofstatements * numberofoutcomes$$

When we refer to “statement”, for simplicity in this example we are referring to the “if” keyword. So, there are two if statements, each with two outcomes depending on the truthiness of the condition, using the above calculation this would be:

$$N = 2 * 2 = 4$$

We can see that this matches the number of total outcomes in the method “foo”.

It is important to understand that the major difference between NPATH and Cyclomatic Complexity is that NPATH calculates the number of acyclic executions rather than the cyclic executions. More formally, instead of adding based on the control flow graph we simply measure the number of paths in the flow with respect to the source code. Like Cyclomatic Complexity, this could be important to the model as it is possible methods with higher NPATH values may be more brittle than those with lower values.

3.3.3 Non-Commenting Source Statements (NCSS)

Non-Commenting Source Statements are rather self-describing. This represents the number of statements in source code that is an actual language statement, it does not include comments, semicolons, or empty statements. Additionally, if a single statement is split onto multiple lines, then this will only be counted as 1. This is useful for calculating the real weight of source code, as often we write programs in a certain way to make it easier to read in the future. However, this can skew the number of statements which are executed. For more detail, a software metric suite which uses this was developed by Clemens Lee called JavaNCSS [8]. As we are effectively counting the number of "real" code statements, this may be important to the model as the more lines of code that could be modified, could either increase or decrease the success rate of a genetic edit being applied.

3.3.4 Metrics that utilise Method Length

Additional metrics can be provided by normalising the metrics by the method length - the number of lines in a method. These are calculated by taking the value of a metric like Non-Commenting Source Statements and dividing it by the number of lines in the method. For example, let's define the value of Non-Commenting Source Statements as "NCSS" and the number of lines in the method as "Length". We will call the result "P".

$$P = NCSS \div Length$$

Now let's set NCSS = 2 and the length = 6

$$P = 2 \div 6 = 0.33..$$

The value we end up as can be looked at as the percentage of lines in the method with are NCSS. So, in this case approx. 33.3% of the method lines are NCSS. These metrics will be abbreviated to things like ncss/length, cyc/length and npath/length. These could be important to the model as they represent a proportion of source code which is considered one of the following metrics. The same rules as described above for the individual metrics applies here.

3.3.5 Metrics that utilise Definition-use Distance

Metrics can utilise the definition distance between where certain elements of source code are defined and where they are used. For instance, the metric MedDefUseDist is the distance between where the method signature is declared and where it is used. Let’s consider the following example source code, assuming that these two methods appear one after the other as shown below:

```
public void methodA() {
    int a = 2;
   // do work
}


public void methodB() {
  int b = 3;
  String a = “”;
  methodA();
}
```

The Definition-use Distance for methodA() would be 9, as it is defined and then used 9 lines later in methodB(). This could be important when use as a feature to models as the more distance between the definition and its use could increase or decrease the success rate of the edit which is being applied.

This same kind of metric can be applied to variables. The metric AveUseDefDist measures the average distance between where a variable is declared and when it is used.

3.4 The CRISP-DM Framework

This section will briefly introduce the CRISP-DM (**Cross-Industry Standard Process for Data Mining)** framework. [18] This methodology will be used and modelled for the implementation of the machine learning models.

CRISP-DM is a framework which models the process of data mining or machine learning and can be used to guide the process of carrying out a machine learning project. This is split into six main stages, which are as follows:

1. Business Understanding – Setting the objectives from a business perspective, producing a plan for the project, and laying out business success criteria.
2. Data Understanding – Exploring the data, understanding relationships between the data, and verifying the quality of the data.
3. Data Preparation – Cleaning the data, removing duplicates, preparing the data to be used.
4. Modelling – Selecting modelling techniques, building models, and assessing their accuracy
5. Evaluation – Evaluate the success of the models against the business objectives that were laid out.
6. Deployment – Take evaluated results and determine how best to deploy them, review the project.

It is likely that this project will not follow this process 100% strictly, but loosely follow these stages in an iterative and agile process with the attempt of building on previous attempts to create accurate predictive models. An illustration is shown in figure 7 below.

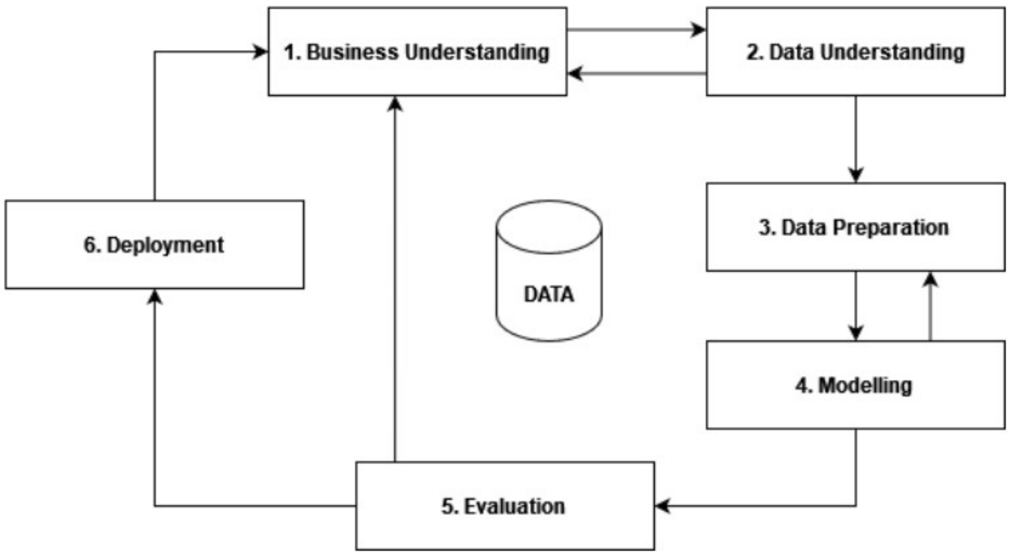


Figure 7. Illustration of CRISP-DM Framework

3.5 Chosen Machine Learning Framework and Alternatives

This section will discuss the machine learning framework that was chosen for implementation in this project. The next few sections will briefly describe the options that were considered and then conclude with a summary, describing the option that was chosen and why, as well as why the other options were not considered.

3.5.1 Sklearn and Python

Sklearn (Scikit-learn) [19] is a popular machine learning framework which uses the Python language. It is open source and built on several other popular Python libraries such as NumPy, SciPy and matplotlib. It is easily integrated with Jupyter Notebooks which is a common choice for data scientists that allows code and documentation to be combined seamlessly. The documentation is vast and contains a lot of useful tutorials and examples for learning the library. Sklearn also has a reasonably active community on their Github, and it is regularly updated with new features. The Python language is quite accessible for learning and thus makes this good all-around option.

3.5.2 R and Caret

Caret [20] is a machine learning package for the R programming language [21]. R is a programming language that was designed to be used for statistical computing and graphics. It has features to analyse and graph data which are built into the core of the language. Caret provides pre-processing, visualisation, data splitting and a variety of models which can be trained and tuned. The documentation for Caret contains a lot of information on how to get started and provides good examples. This requires a foundation in R, which is known to be trickier to learn as it is quite substantially different in syntax to other programming languages.

3.5.3 Java and DeepLearning4J

DeepLearning4J [22] is a framework and collection of tools for running machine learning on the Java Virtual Machine (JVM). This uses Java as the base language but will interoperate with some Python libraries for execution, and for training and testing models. This is useful for having an existing knowledge in Java, as it removes the need to learn a new language in order to carry out machine learning tasks. The documentation is quite verbose and covers many different topics regarding the language.

3.5.4 Conclusion

For this project, Sklearn [19] and Python was chosen as the machine learning framework to be used in this project. As mentioned in section 3.5.2, R is a much more difficult language to adjust to using because of its significantly different syntax and although this language is more geared towards machine and statistical analysis it would make implementation a lot more difficult. DeepLearning4J uses Java but piggybacks off a lot of Python infrastructure, which means there still may be a requirement to learn how those work outside of the main suite of tools. There also can be some overhead when using Java compared to other languages. Sklearn not only has a very active support community and is regularly updated, it also is built to integrate with Jupyter Notebooks. This is one of the main reasons it was chosen, as these notebooks can contain Python source code and text. Finally, Python is a much easier language to transition to and many of the other libraries which are included with Sklearn are also well maintained and have good functionality for their use cases.

4 Feature Construction with JavaParser

The following sections will discuss the tool JavaParser and how it was used to construct simple features to be used as inputs to the model.

4.1 JavaParser

JavaParser [23] is a Java Language Parsing tool which gives the ability to interact with java source files as an Abstract Syntax Tree. An Abstract Syntax tree represents source code in a tree like structure, where statements in the language are broke into smaller parts that form the nodes of the trees and branches between those trees allow the ability to "walk" to different nodes in the tree. This allows for powerful source code analysis and parsing as the entire contents of the source file can be looked at independently for different purposes. JavaParser solely provides this functionality to interpret a Java Source File as an AST, any other use cases, such as computing metrics of a source file are left up to the user of the library.

JavaParser breaks up a java source file into its key components, with a CompilationUnit being the root of the source file. We can visit specific nodes such as method declarations or even retrieve all the method names and parameters that are contained within the source file. This will allow more detailed analysis of source code; this is shown in figure 8 below.

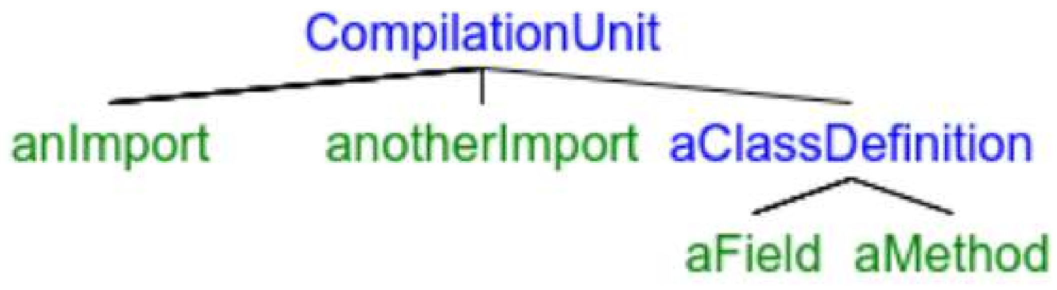


Figure 8. Example of JavaParser AST [24]

In this project, JavaParser was used to analyse the java source files contained in the open-source projects which make up the dataset and construct metrics which will be used as features to the model. This use of JavaParser is non-standard and as stated above, this functionality is required to be implemented separately. A simple Java program was created to house the feature construction and calculate these new metrics over the open-source projects contained in the dataset. The GI dataset used is defined in more detail in section 5.2.1.

The next few sections will give a high-level overview of the feature or metric construction that was implemented for this project. The actual code required to write these was relatively simple, therefore the focus will be on how these were constructed and how they would be applied to source code.

4.2 Using JavaParser to Construct New Metrics

The following sections will describe and illustrate how JavaParser was used to construct custom metrics to evaluate source code and be used as features. These metrics were computed at the method level as the dataset focuses on genetic edits which were made to methods.

4.2.1 Definition of Metrics

This section will describe the definition of the metrics that were constructed for this project.

Before any implementation could be started regarding constructing custom Metrics with JavaParser, a definition of what these metrics would calculate was required. For the sake of this project, these were kept to very simple cumulative metrics. Four of these were defined and implemented, they are as follows:

1. Number Nested – The number of nested statements
2. Number Surfaced – The number of surface statements
3. Total Iterative – The total number of iterative statements
4. Total Conditional – The total number of conditional statements

As we do not want these newly created metrics to calculate the same things as other metrics which were provided by Checkstyle [25], a rule was put in place for what designates a surface level statement and a nested level statement. A nested statement is any statement whose parent is not the method declaration, and a surface statement is any statement whose parent is the method declaration. The total number of iterative and conditional statements were found by calculating the sum of the number of nested and surface statements which met the criteria for either of these.

The number of nested and surface statements were made up of all standard Java language specific expressions.

4.2.2 Example of Applying Metrics to Source Code

Let’s look at a worked example to see the output of these metrics given a very simple method in Java:

Surface

Nested

```
public void foo() {
    if(true) {
        for(;;) {
        }
    }
   switch(a) {
      if(true) {
      }
   }
}
```

This method is obviously very simplistic and production code methods would likely contain many more lines. However, by running the metric computation on the method foo we would get the following output, show in table 1 (note this only shows elements present in the method, non-present metrics would get a value of 0 as expected):

surfaceIfs	nestedIfs	surfaceSwitches	nestedFors	iterativeStmts	conditionalStmts
1	1	1	1	1	3

Table 1. Count of Metrics in Example

As can be seen, we walk the AST to find the number of surface and nested statements for each statement type, and then add together the statements that fit the criteria for iterative or conditional. In this case we have only one iterative statement but three conditional statements in total.

To help illustrate how this would look in an AST form, a simplified visualisation of this is provided below in figure 9. This does not include the semantics that the tree outputted by JavaParser would produce, to reduce complexity.

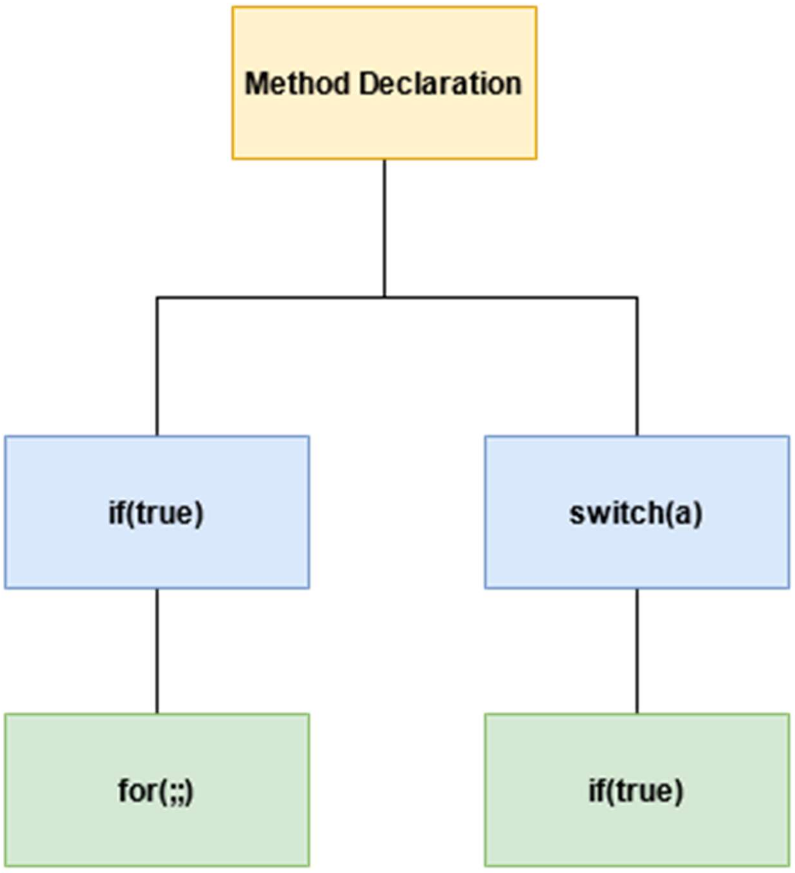


Figure 9. Simplified AST of Foo Method

As can be seen by visualising it in this way, we can see from an abstract point of view how we would obtain these values by walking the AST to the appropriate positions and returning the number of nodes which match.

4.2.3 How these differ from Cyclomatic Complexity and NPATH

At first glance these metrics may seem to calculate very similar if not identical metrics to Cyclomatic Complexity and NPATH. As stated in the definition, a rule was put in place which attempts to prevent this. In both Cyclomatic Complexity and NPATH we are counting paths and control flow through source code. These metrics are not concerned with control flow or paths throughout the source code, instead these simply calculate the occurrence of certain statements or kinds of statements that appear in a method. The idea was to investigate whether these simple metrics would have any bearing on the success rate of a genetic edit, and whether it would be possible to use these metrics to help guide GI of Software and determine *amenable* regions.

5 Design and Implementation

The following sections will cover the design stages, modelling loosely after the CRISP-DM framework. As stated in section 3.4, due to the nature of this project the stages will not necessarily strictly reflect the steps carried out at each stage. Results and important implementation details will be covered at high level, and finally an evaluation of the final product will be conducted.

The design stages will be illustrated as an overview of the project. Although there were two major cycles made through the process, the main factor that changed was the features that were investigated. The features that were constructed for the second iteration were discussed in the last chapter, and this is to ensure that the results can be presented all in one place.

5.1 Business Understanding

The business understanding for this was largely covered in the problem description of this project. Please see section 3.1-3.2 for more detail. As a recap, the goal is to use machine learning to help guide GI of software, by predicting regions of source code which are most *amenable* to edits. The next few sections will talk in more detail about the methodology used at each stage of the CRISP-DM process [18] that is relevant to this project.

5.2 Data Understanding

The following sections will cover the data understanding that was performed for this project.

5.2.1 The GI Dataset

The beginning step for understanding the data was to explore any important relationships between the features.

Before proceeding to the rest of the data understanding, it is important to understand the dataset itself in more detail, and what it is composed of. This was briefly covered in Section 3.1 but will be described in more detail here.

Over 2000 thousand genetic edits were applied uniformly at random to 8 open-source java projects. The GI toolkit Gin does not have the capability to generate metrics on source code by itself, so another tool was used for this. The tool that was used to generate the metrics is a Java specific static code analysis tool called Checkstyle. Checkstyle [25] can generate the metrics set out in section 3.3 on a specified piece of source code. The edits were applied first to the open-source projects and then Checkstyle was used to obtain the metrics for the source code in those projects. These were then combined to create the dataset which is used for this project. This initial dataset was provided by my supervisor and generated as part of experiments for a paper. [26]

Additional features were created using JavaParser in the second cycle of this implementation which was covered previously in Chapter 4.

5.2.2 The relationship between Compile and Pass features in GI Dataset

The first important thing of note that was explored in the dataset for this project, is the relationship between rates related to Lines or Statements which have passed and those that have compiled. There is a high positive correlation between passed features and compile features. If we plot the relationship between the Proportion of Lines Compiled and the Proportion of Lines Passed (two of the proposed targets) using a scatterplot we can see this more clearly. This correlation is shown in figure 10 below.

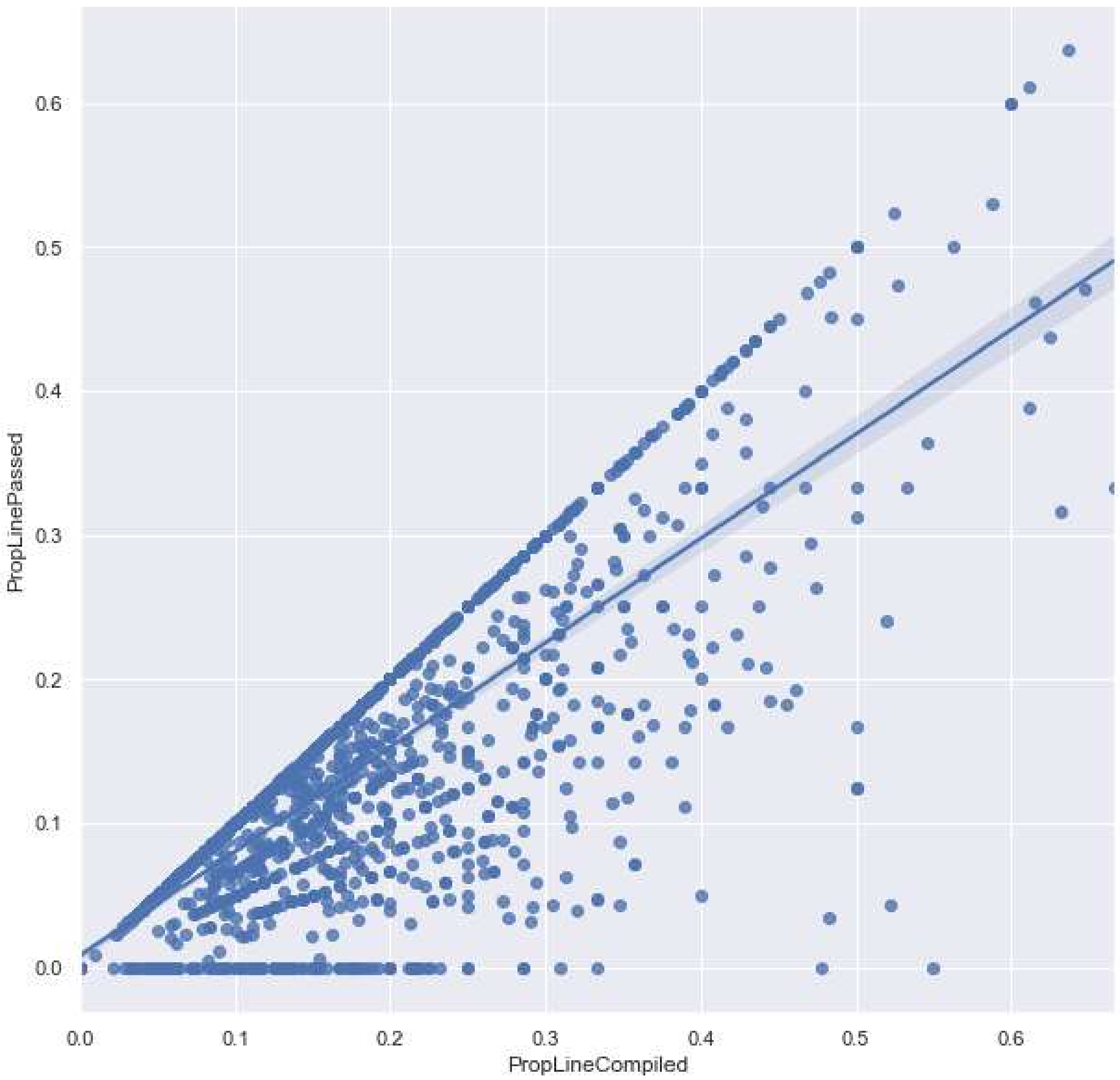


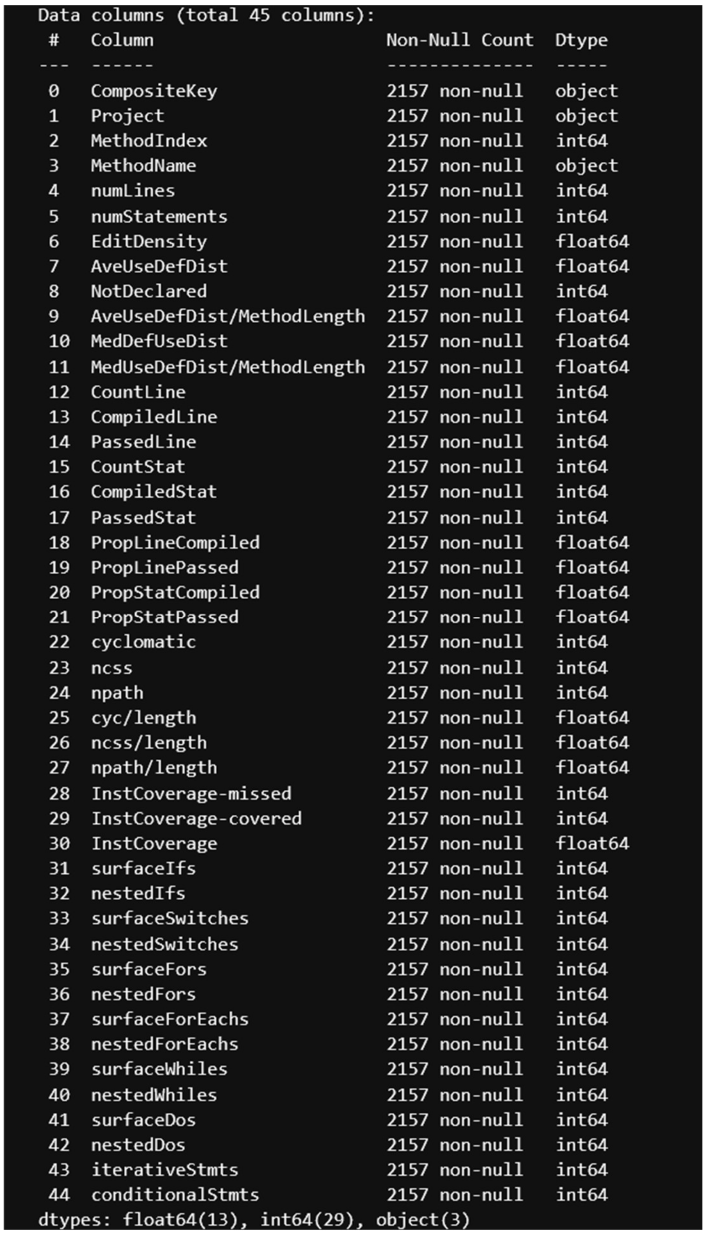
Figure 10. Scatterplot Relationship between PropLinePassed & PropLineCompiled

This visualisation shows the high positive correlation between passed features and compiled features. (This also works the same way in reverse) This is an important factor to understand as we are concerned with determining whether features like software metrics can be used to predict *amenable* regions. These high collinearity features will likely make the model seem more accurate than it is, and skew results. These will still be investigated when selecting features, to show their impact but will be dropped from the final selection. There is a cost to measuring either compile or pass rates, and the purpose of using machine learning is to attempt to avoid that cost.

5.2.3 Analysing and Inspecting the GI Dataset

Before preparing the data to be used in the modelling stage, it is important to analyse and inspect the dataset to gain a better understanding of the makeup of the data, as well as discovering if there are any errors or missing data which will need to be addressed before fitting the dataset to candidate models. If these are not considered, then it could skew the results by either overestimating or underestimating the accuracy. Another consideration is to identify early what features are not necessary for the model, I.E those which are not useful for making predictions.

The columns that may form features, as well as their type is shown in the figure 11 output below.



```
Data columns (total 45 columns):
 #   Column                       Non-Null Count  Dtype  
---  ------                       --------------  -----  
 0   CompositeKey                 2157 non-null   object 
 1   Project                      2157 non-null   object 
 2   MethodIndex                  2157 non-null   int64  
 3   MethodName                   2157 non-null   object 
 4   numLines                     2157 non-null   int64  
 5   numStatements                2157 non-null   int64  
 6   EditDensity                  2157 non-null   float64
 7   AveUseDefDist                2157 non-null   float64
 8   NotDeclared                  2157 non-null   int64  
 9   AveUseDefDist/MethodLength   2157 non-null   float64
 10  MedDefUseDist                2157 non-null   float64
 11  MedUseDefDist/MethodLength   2157 non-null   float64
 12  CountLine                    2157 non-null   int64  
 13  CompiledLine                 2157 non-null   int64  
 14  PassedLine                   2157 non-null   int64  
 15  CountStat                    2157 non-null   int64  
 16  CompiledStat                 2157 non-null   int64  
 17  PassedStat                   2157 non-null   int64  
 18  PropLineCompiled             2157 non-null   float64
 19  PropLinePassed               2157 non-null   float64
 20  PropStatCompiled             2157 non-null   float64
 21  PropStatPassed               2157 non-null   float64
 22  cyclomatic                   2157 non-null   int64  
 23  ncss                         2157 non-null   int64  
 24  npath                        2157 non-null   int64  
 25  cyc/length                   2157 non-null   float64
 26  ncss/length                  2157 non-null   float64
 27  npath/length                 2157 non-null   float64
 28  InstCoverage-missed          2157 non-null   int64  
 29  InstCoverage-covered         2157 non-null   int64  
 30  InstCoverage                 2157 non-null   float64
 31  surfaceIfs                   2157 non-null   int64  
 32  nestedIfs                    2157 non-null   int64  
 33  surfaceSwitches              2157 non-null   int64  
 34  nestedSwitches               2157 non-null   int64  
 35  surfaceFors                  2157 non-null   int64  
 36  nestedFors                   2157 non-null   int64  
 37  surfaceForEachs              2157 non-null   int64  
 38  nestedForEachs               2157 non-null   int64  
 39  surfaceWhiles                2157 non-null   int64  
 40  nestedWhiles                 2157 non-null   int64  
 41  surfaceDos                   2157 non-null   int64  
 42  nestedDos                    2157 non-null   int64  
 43  iterativeStmts               2157 non-null   int64  
 44  conditionalStmts             2157 non-null   int64  
dtypes: float64(13), int64(29), object(3)
```

Figure 11. Output showing GI Dataset Columns and Types

For this project, the two major issues that were identified when inspecting the dataset was that two features npath and AveUseDefDist had values of -1, which are not sensible values for those features. This indicated that there was missing data for these, and that they would need to be addressed in the Data Preparation stage.

The features CompositeKey, Project and MethodName were all identified as not relevant to the model, because this is a regression problem, they are not relevant to the model and are also unlikely to have any real impact on the accuracy. Additionally, MethodIndex was also identified as not relevant as it was determined that the index of the method would likely not be a useful predictor.

5.3 Data Preparation

This section will detail the Data Preparation stage, including the cleaning and filtering of data and any changes made to ensure that the dataset was ready for modelling.

5.3.1 Cleaning & Filtering of Data

Before beginning the modelling process, it is important to make sure that the dataset is prepared, and any errors are addressed. It is also essential to ensure that irrelevant data is removed so that the analysis is focused on the core objectives of that the model is being created to achieve.

There were no duplicates or null values found in the dataset, so there was no need to remove those.

In the previous section, it was identified that the features npath and AveUseDefDist had missing data; values of -1. For npath it was difficult to determine the root cause of this missing data, it is likely an unidentified issue with the way that Checkstyle [25] calculated this metric at runtime. To solve this, going forward a filter will be applied to the dataset that only contains entries with a npath value > -1.

For AveUseDefDist, after some investigation of the methods contained in the dataset it was found that -1 seemed to appear when a variable was declared in a loop. Checkstyle [25] treats this as if the variable had not been declared because it interprets these as if they have a definition distance of 0, which results in the -1 outputs. For example, take the following code snippet:

```
for(String item : list) {
        //Do work
}
```

Checkstyle would treat this as having a definition distance of 0. There were two ways that this could be addressed, the first is to change all -1s to 0 to match what it was originally interpreted as or introduce a new feature which accounts for this. The later was chosen and the feature was named "NotDeclared", this was set to a value of 1 if AveUseDefDist was -1 and 0 if it was not.

Finally, we want to make sure that we are investigating methods which have substance, that is a reasonable number of lines in the method. So, a filter was applied for the dataset to only include methods which have > 10 lines.

The original number of entries was 2157, after preparing the data, the working set was reduced to 925.

5.4 Modelling

This section will detail the modelling stage, including the modelling technique, the models that were selected and how they were improved using hyperparameter tuning.

5.4.1 Modelling Technique Overview

This project will use a Supervised Learning modelling technique. This involves using a training set which will teach the proposed models to produce the desired output. For this project, this will take the form of a regression problem. The focus for this is on understanding relationships between independent variables (selected features) and dependent variable(s) (target features). The outputs of these are a numerical value, and as stated in the previous sections, there are four targets which were investigated, and these are numerical values which represent the proportion of compile and pass features. The dataset was split between 70% for the training set and 30% for the test set.

5.4.2 Feature Importance

In this project, the model that is the most relevant consists of just the features which are metrics as inputs, as this is part of the objectives to identify a connection between these and *amenable* regions. However, to investigate how the features affect the model when they are added or removed, time was spent on examining the feature importance and comparing the accuracy at each stage with respect to the targets.

In Sklearn [19] we can interrogate the feature importance of a model and plot it as a bar graph, showing the ranking of what features are important to the model. The features which are most important are normally those which contribute most to the accuracy of the model.

There were four targets considered in this project but the process for acquiring feature importance is the same. This will use the target PropLinePassed and the graphs which were produce for the first iteration cycle.

To get access to the feature importance values, we will create a temporary default RandomForest model, the training and test sets were split the same as stated in the previous section.

First, let’s examine the top 5 important features when we include all features except the target. This is shown in figure 12 below.

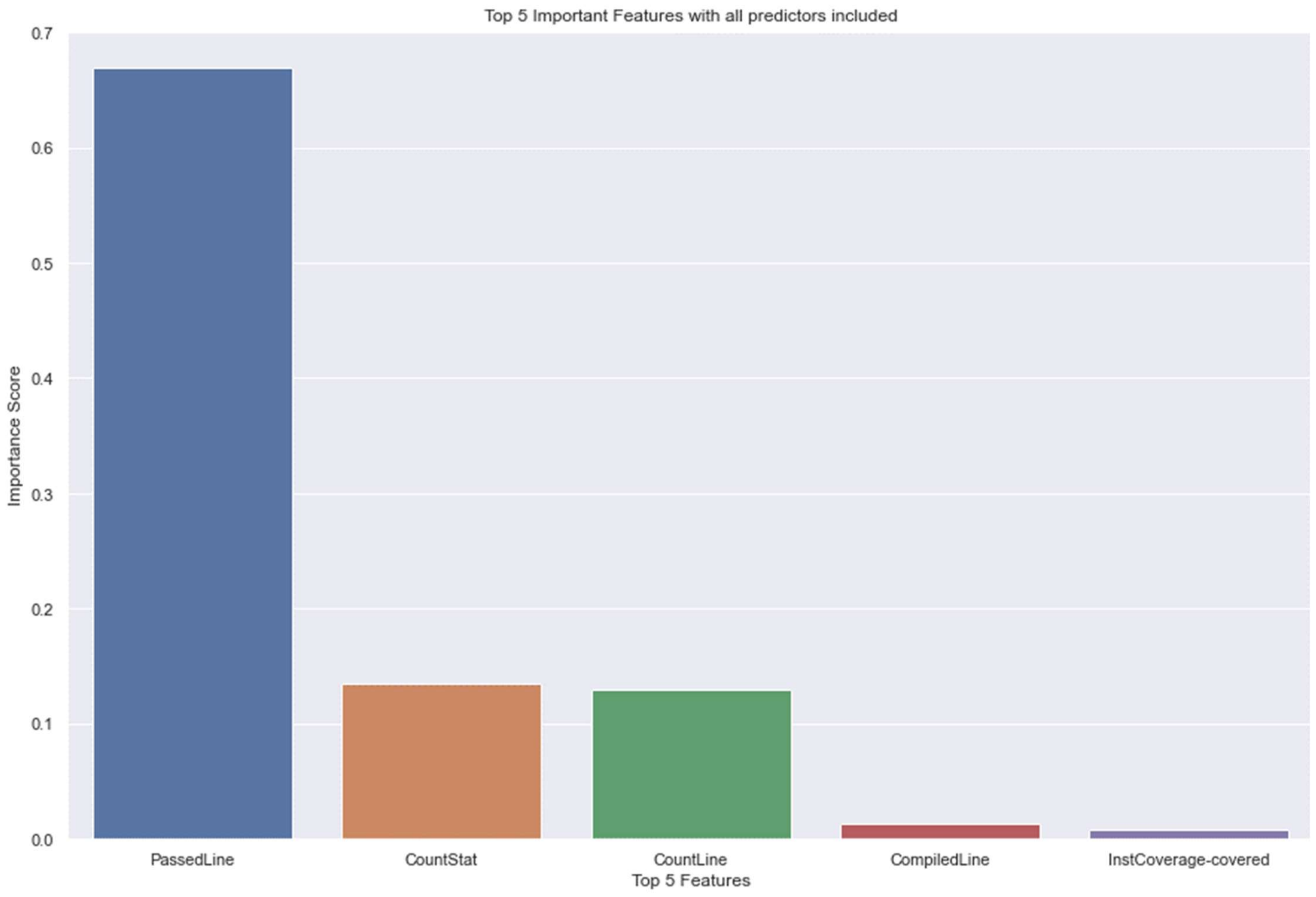


Figure 12. Bar Graph of Top 5 Important features, all predictors included

The R2 value for this model with all features included as input is 95%. As shown in figure 10, the most important feature is PassedLine. PassedLine is just the number of lines which passed, and this feature makes up one part of the target PropLinePassed. This feature therefore highly inflates the accuracy of the model.

Now we will remove this feature and examine the impact it has on the model, as shown in figure 13 below.

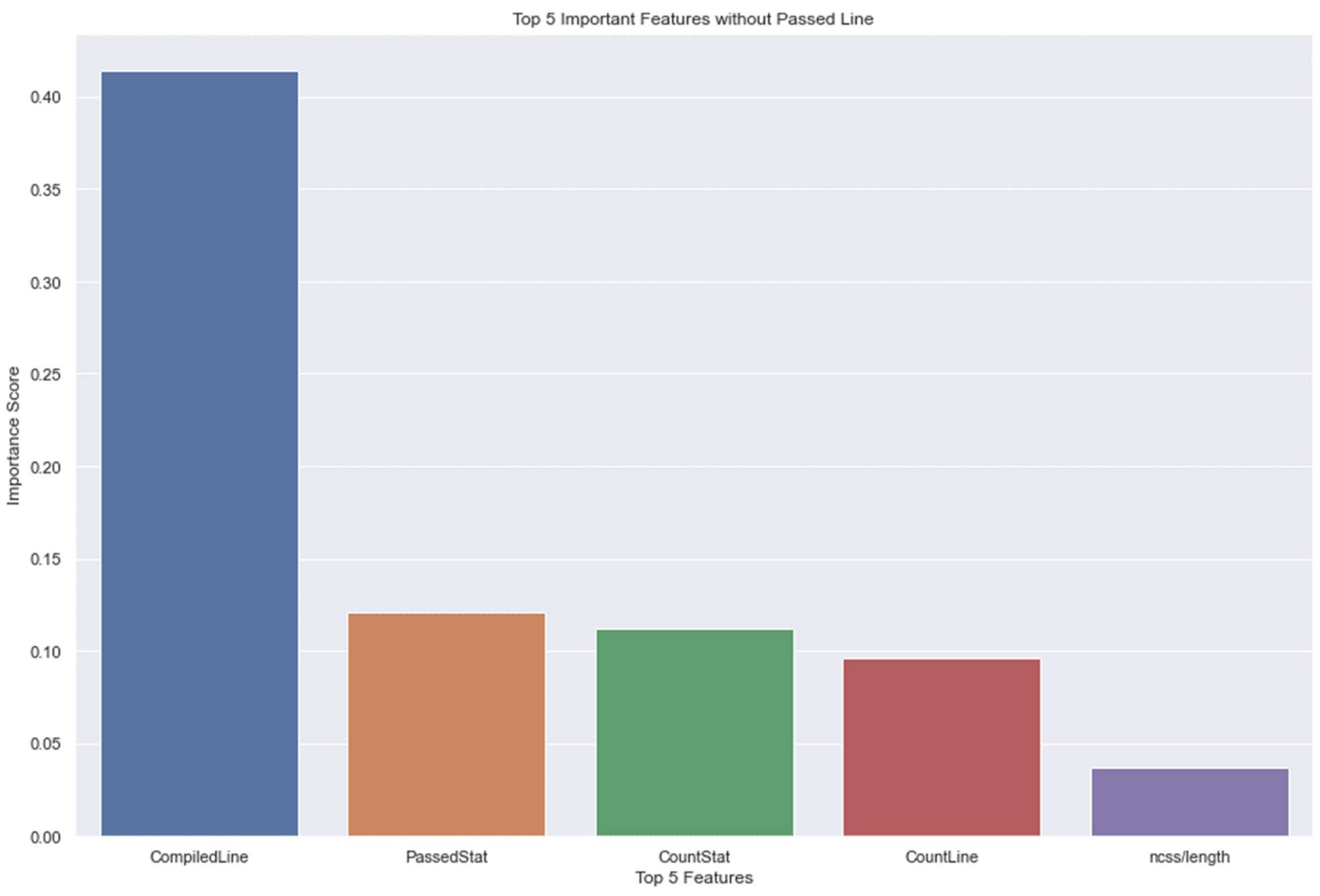


Figure 13. Bar Graph of Top 5 Important features, without Passed Line

In this example, we see the problem of high collinearity between pass features and compile that was discussed in Section 5.2.2 demonstrated by the fact that the CompiledLine feature, which is the number of lines that compiled is the most important feature. The R2 score without PassedLine drops to 67%, which is still quite accurate, but this is being skewed by the inclusion of CompiledLine.

We will now look at the important features for a model which is built with only the metrics from Checkstyle [25] as shown in figure 14 below.

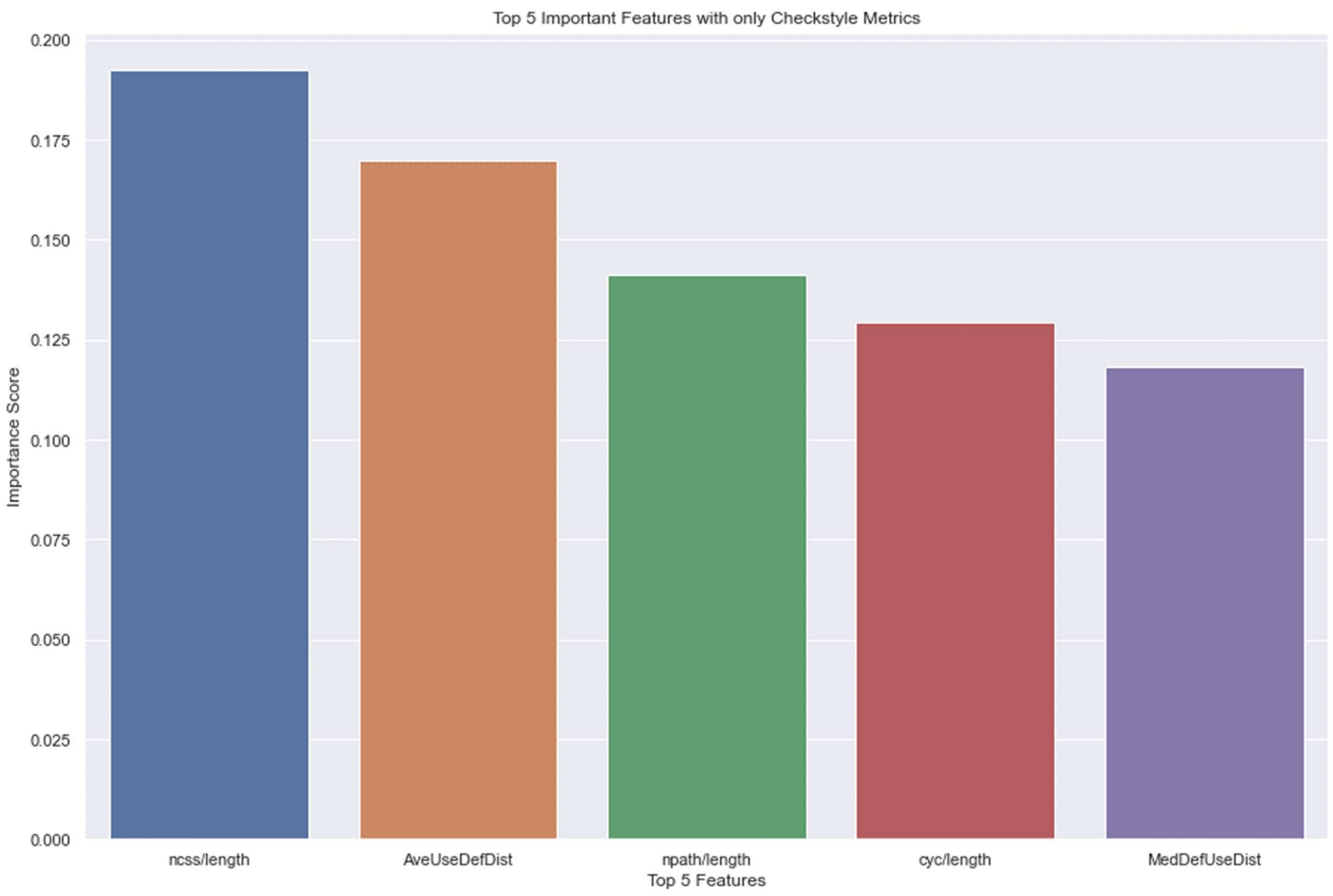


Figure 14. Bar Graph of Top 5 Important features, with only Checkstyle Metrics

For the metrics, the most important feature is NCSS (Non-Commenting Source Statements) / the method length. Which highlights that this could be a feature that could be investigated further. However, the R2 score has dropped to only 2.7% which suggests that there may not be a strong connection between the metrics and the data and that it will be difficult to produce accurate models with respect to this target. (There is an additional graph provided for the top 5 important JavaParser features in the appendix)

5.4.3 Model Selection

To ensure that a fruitful investigation is performed, a variety of models will be selected and used for each of the proposed targets.

For this project, four different model types were selected to be used and evaluated. These were:

1. Ridge Regression
2. KNN (K Nearest Neighbours)
3. RandomForest
4. MLPRegressor

To help find a predictive model, these different types were selected to ensure that a comprehensive investigation can be made.

5.4.4 Hyperparameters & Tuning

In this project, it was important to ensure that good models were built. Even in cases where models may produce a low accuracy, we can improve this by tuning the models hyperparameters. A hyperparameter can simply be defined as a parameter which is used to control the learning process of the model. By tuning these models, we can improve the accuracy. In a regression problem such as this, we can measure a form of accuracy known as R Squared (R2). This value represents how well the model can explain the variance in the dataset. A higher value means that the model will be able to predict more accurately. We can also calculate an adjusted R2 value which will give a more realistic interpretation of the accuracy of the models.

For this project, the four candidate models that were chosen in the previous section were first evaluated without any hyperparameter tuning and then with hyperparameter tuning. This will be covered in more detail in the Evaluation section.

The first steps for tuning these candidate models were to select a series of hyperparameters for each model that were going to be used and pick a search method and validation technique.

The search method that was chosen for tuning the hyperparameters for the selected model was Random Search. This is a search technique that will search a given search space over a set number of iterations, choosing different combination of hyperparameter values at random and choosing a "best fit" each time. The best fit at the end of the iterations will be the combination of hyperparameters which is outputted. This search used a repeating 10 Cross Fold validation technique, which splits the data sample each time into unique groups so that it is evaluated more fairly.

The number of iterations used for this search was set at 800. Table 2 below shows the candidate models, the hyperparameters chosen and the values used for the search space.

Candidate Model	Chosen Hyperparameters	Hyperparameter Values
Ridge	Solver Alpha Fit Intercept Normalise	svd, Cholesky, lsqr, sag log uniform value between 0.00001 and 100 True, False True, False
KNN	Algorithm N Neighbours Weights Leaf Size P	Ball Tree, KD tree, brute 2, 3, 5, 10, 12, 15, 20, 25, 30, 40 Uniform, Distance 10, 30, 60, 80, 90, 120, 140 1, 2, 3
RandomForest	N Estimators Bootstrap Warm Start CCP Alpha	100, 250, 300, 350, 400, 500 True, False True, False log uniform value between 0.00001 and 100
MLPRegressor	Hidden Layer Sizes Activation Alpha Learning Rate Solver	(50, 50, 50), (50, 100, 50), (100, 50, 1) Relu, tanh, logistic log uniform value between 0.00001 and 100 constant, adaptive adam, lbfgs

Table 2. Candidate Models and their Hyperparameters

These models were tuned by performing the random search with respect to the range of values shown above.

5.4.5 Explainability Techniques and their Usefulness

Explainability is concerned with further exploring the importance of features to data points within the model. This is useful for gaining more information about the relationships a given feature has with machine learning models and understanding whether those features positively or negatively impact that predictability. There are many ways we can look at this, we can first look at the coefficients of the model to determine a positive increase or negative decrease between the target and the selected feature.

To illustrate this with relation to this project, we will look at the target PropLinePassed or the percentage of lines which passed their unit tests. This will use a simple LinearRegression Model. For the sake of brevity, we will display the features that were constructed using JavaParser [23] as outlined in the previous chapter, this is shown in Table 3 below.

Feature Name	Coefficient
surfaceIfs	-0.0017
nestedIfs	-0.0014
surfaceSwitches	0.0055
nestedSwitches	-0.0011
surfaceFors	0.0097
nestedFors	0.008
surfaceForEachs	0.007
nestedForEachs	0.004
surfaceWhiles	0.0002
nestedWhiles	-0.0084
surfaceDos	-0.0453
nestedDos	0.0177
iterativeStmts	-0.0072
conditionalStmts	0.0013

Table 3. JavaParser features and their Coefficients

Positive coefficients = An increase in the selected feature results in an increase in the target feature

Negative coefficients = An increase in the selected features results in a decrease in the target feature

This only shows us part of the picture when we are looking at these features with respect to the target. The model used is a LinearRegression model, which as the name suggests predicts in a linear way, I.E searching for a straight line. But there is a chance the relationship may be more complicated than that if we fit a non-linear model.

To visualise this difference, we can use partial dependence [27]. This is where we look at a smaller input set of the features we are interested in, and we can gauge both the relationship and how the selected feature affects the model's predictability. Let's select the feature iterativeStms and we will set our smaller set of features at 100. We can see the coefficient for this feature was a small negative output of -0.0072. We'll first use a partial dependency plot with the same LinearRegression model, as shown in figure 15.

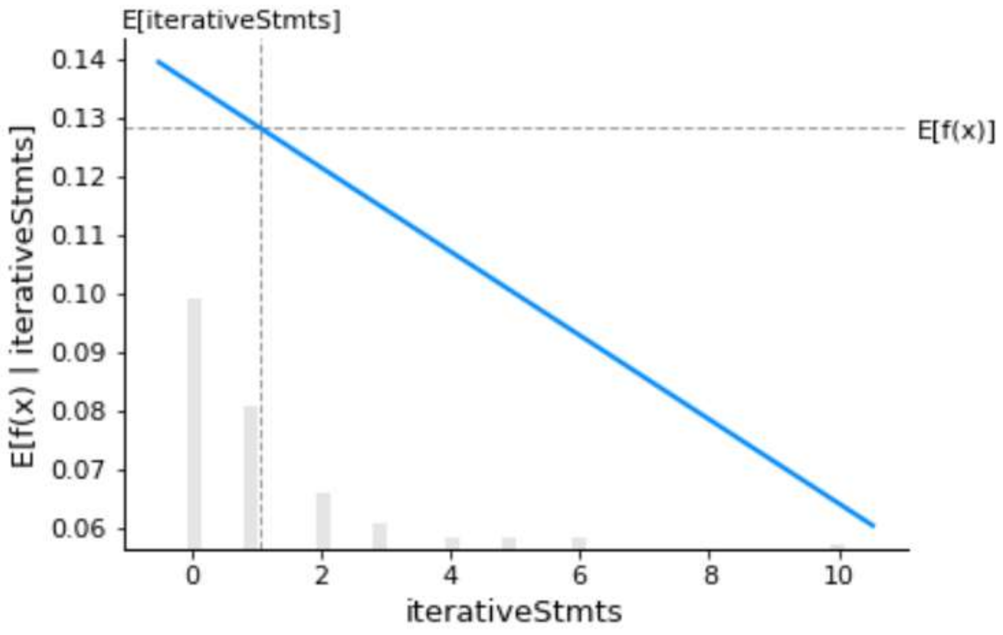


Figure 15. LinearRegression Partial Dependence Plot For iterativeStmts

E[f(x)] is the average prediction of the model, and we can see that this model shows a linear negative trend for surface level if statements with respect to the model. Next, we will instead produce a partial dependency plot with a non-linear RandomForest model, as shown in figure 16 below.

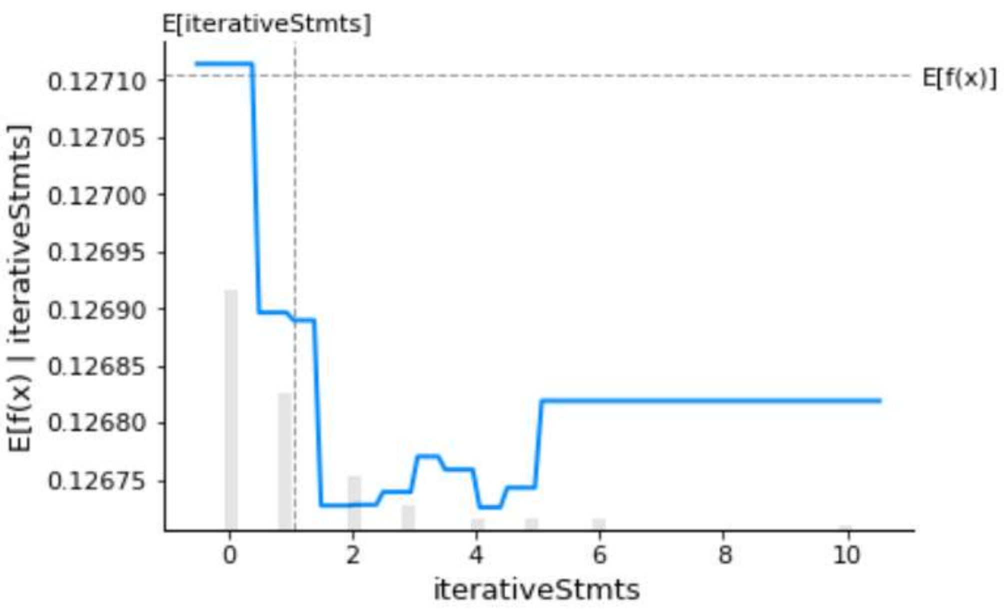


Figure 16. RandomForest Partial Dependence Plot

As can be seen, although the above LinearRegression model showed a simple negative trend, when we look at this relationship with a RandomForest model we can see it is much more complicated. The trend does go towards the negative, but it peaks at certain points and levels out towards the end. This is useful for this project in demonstrating that even simple features may have more complex relationships and require further investigation.

5.5 Evaluation

This section will cover the evaluation stage. This will divert slightly from the CRISP-DM stage, in that it will include the results from the two iterations that were carried out. The first covering the initial metrics from Checkstyle [25], and the second when features were constructed using JavaParser [23], to keep the results in one place. This section will also evaluate these results and the product, as well as drawing conclusions from them.

5.5.1 Results

The results for the models which are presented in this section were tuned using the hyperparameters and rules that were set out in Section 5.4.4. Although briefly mentioned in previous sections, the measure that we will be using to assess how accurate a model is by using the R2 value. This represents how well the model explains the variance in the dataset. The closer the value is to 1 the more accurate the model is. Additionally, we will also use the Adjusted R2 value to compliment this. It is possible that the R Squared value will overestimate how accurate the model, and this accounts for this and provides a more realistic value for accuracy.

The tables provided below show the models and the respective R Squared and Adjusted R Squared values for each target. The most accurate model will be highlighted in green, and the worst will be highlighted in red.

We will begin by looking at the results for the model which used only the Checkstyle [25] metrics as inputs. This was the first iteration that was carried out for this project, as shown in table 4.

Model	PropLinePassed	PropStatPassed	PropLineCompiled	PropStatCompiled
Ridge	R2: 0.04508	R2: 0.00868	R2: 0.08344	R2: 0.00058
	Adj R2: 0.03569	Adj R2: -0.00107	Adj R2: 0.07443	Adj R2: -0.00925
KNN	R2: 0.00452	R2: 0.03733	R2: 0.04047	R2: 0.02314
	Adj R2: -0.00527	Adj R2: 0.02757	Adj R2: 0.03103	Adj R2: 0.01353
RandomForest	R2: 0.06307	R2: -0.00551	R2: 0.08840	R2: 0.04960
	Adj R2: 0.05385	Adj R2: -0.01570	Adj R2: 0.07944	Adj R2: 0.04025
MLPRegressor	R2: 0.02516	R2: 0.00757	R2: -0.00521	R2: -0.00685
	Adj R2: 0.01557	Adj R2: -0.00219	Adj R2: -0.01509	Adj R2: -0.01676

Table 4. Results for First Iteration models with Checkstyle

The RandomForest model was the most consistently accurate model across the targets, whereas the MLPRegressor performed the worst. The highest R Squared Value was 0.08840 or 8.84% with respect to PropLineCompiled. The lowest was -0.00685 or -0.68% and this was with respect to PropStatCompiled.

We will now look at the results for the model which used the custom metrics created using JavaParser, as discussed in Chapter 4. This was the second iteration was carried out for this project, as show in table 5.

Model	PropLinePassed	PropStatPassed	PropLineCompiled	PropStatCompiled
Ridge	R2: 0.01124	R2: -0.01430	R2: -0.00701	R2: -0.03390
	Adj R2: 0.00042	Adj R2: -0.02540	Adj R2: -0.01803	Adj R2: -0.04521
KNN	R2: 0.01041	R2: 0.02972	R2: 0.03705	R2: 0.01732
	Adj R2: -0.00042	Adj R2: 0.01910	Adj R2: 0.02651	Adj R2: 0.00657
RandomForest	R2: 0.00445	R2: 0.07215	R2: 0.03007	R2: 0.04116
	Adj R2: -0.00644	Adj R2: 0.06199	Adj R2: 0.01945	Adj R2: 0.03067
MLPRegressor	R2: 0.04713	R2: 0.05907	R2: 0.07508	R2: 0.01112
	Adj R2: 0.03776	Adj R2: 0.04982	Adj R2: 0.06598	Adj R2: 0.00139

Table 5. Results for Second Iteration models with JavaParser

RandomForest and MLPRegressor were the most accurate models across the targets, whereas Ridge performed the worst. The highest R Squared value was 0.07508 or 7.5% with respect to PropLineCompiled. The lowest was -0.03390 or 3.39% with respect to PropStatCompiled.

5.5.2 Evaluation and Conclusions from Results

There are a few conclusions that can be drawn from both iterations that were performed in this project. Regarding the first iteration, which used the Checkstyle [25] metrics, this did not produce very accurate models with the highest only being slightly above 8%.

Since these do not appear to contribute significantly to the model, and the accuracy is too low to be useful it is not possible to confidently say there is a connection between software metrics and *amenable* regions of source code. However, this is only viewing the results in isolation, it is of course possible a more comprehensive investigation using machine learning could yield more accurate models.

For the second iteration, after attempting to build models with the Checkstyle metrics the idea was to use JavaParser [23] to create custom simple metrics to see if these would produce accurate models. However, for the metrics that were produced using JavaParser these did not produce accurate models with the highest being around 7.5% which again, is not accurate enough to be useful.

The results found show that the best models produced in the first iteration using Checkstyle perform slightly better on average than the models using JavaParser. However, this was the first time that JavaParser has been used in this way and the metrics that were created were very simple. So, it is possible that more advanced metrics could be created, and they may produce more accurate models.

The most accurate models come when attempting to predict the proportion of lines compiled as the target. It is possible that this could mean predicting the compile rate of lines of code is easier, but with the accuracy numbers as low as they are this cannot be definitively proven from these results.

These results once again reaffirm the difficulty in finding regions of source code to target GI edits, and that the search for the most *amenable* regions is one that will likely require further investigation.

6 Conclusion

6.1 Summary

The following section will contain a summary of what was achieved, this is repeated from the achievements specified in the Abstract and section 1.3 for convenience.

In this project, several features were successfully investigated with respect to predicting amenity for four targets which are based on the success of potential edits. The conclusions found by visualising, performing detailed analysis of the existing input metrics, and investigating the effect of creating custom input features and attempting to build predictive models using these features has highlighted many useful areas for future research. Although this project did not manage to produce accurate models that were capable of accurately predicting editable regions, it was able to reaffirm that this kind of prediction is inherently difficult but is still worth investigating further.

This project was one of the first to use the tool JavaParser to compute custom metrics from source code and use them as input to machine learning models of this kind.

During this project, from a personal standpoint I started with no knowledge of Genetic Improvement of Software and through this I have learned a lot about the research area. I have found that I have developed an even greater interest in the development of this area and will continue to learn more outside of this project. Additionally, I also had the opportunity to apply Machine Learning techniques to a real-world problem for the first time, in which I learned a lot about this as well.

6.2 Evaluation of Project Achievements

The following section will contain a critical evaluation of what was achieved against the project objectives which were specified in section 1.2**.**

The objectives will be repeated here for convenience with a summary of their outcomes following them.

1. **Identify features which are critical that are suitable for predicting the performance of GI edits. Particularly, attempt to identity a connection with Software Metrics**:
    a. There were many features which were included in the dataset which had a high collinearity with the targets that were chosen; the proportion of line and statements compiled and passed. The features which came out as being the strongest predictors of the targets tended to be features which overlapped in some way. For example, when predicting the compile rate, features related to the pass rate will be the strongest and vice versa.
    b. The Software Metrics computed by Checkstyle that were used as features had smaller relations to the targets and low importance to the model(s). The majority of these did have higher yield when looking at the line level as opposed to the statement level.
    c. Similarly, the features that were constructed by JavaParser had small relations to the target and low importance to the model(s). These were more consistent across line and statement levels.
    d. In conclusion, although there were many features identified most of them were overlaps or targets. This failed to establish a connection between Software Metrics and the success rate of applied GI edits.
2. **Compare, evaluate, and determine the best Machine Learning model for predicting regions of code to edit using identified features:**
    a. Following the CRISP-DM frameworks model evaluation stage, four different model types were chosen to be evaluated with respect to the chosen features. Those features being Software Metrics computed over the projects used in the dataset.
    b. Hyperparameters were selected for each of these models and then tuned by creating a search space and using a random search technique to improve the model's accuracy.
    c. In conclusion, although the accuracy was not high, a best fitting model was chosen for each target.
3. **Train and test the model from (2) using data generated from Gin, incorporating a range of open-source programming projects to increase generality:**
    a. The model(s) were trained and tested using the dataset, which was provided, and included 8 open-source Java Projects.
    b. Due to time constraints, it was not possible to generate edits for more open-source projects and collect more data.
    c. In conclusion, the models were trained and tested with the dataset as is.
4. **Integrate the model from (2) within the GI toolkit Gin, so that Gin's edits can be guided to the most *amenable* regions of source code. Test experimentally to determine whether this approach improves Gin's efficiency:**
    a. The models produced were not fruitful or accurate enough to be able to test their efficacy.
    b. Therefore, this objective was not met as the highest model accuracy was 8.84% which is not good enough to be reliable or useful.

Overall, when looking at results in isolation. It is rather disappointing that I was unable to get to a stage where I was able to test the results of the Machine Learning models. It was hoped at the beginning of the project I would be able to produce at least some kind of meaningful data even with the dataset as is. However, this was not the case and at most this project can point to the extreme difficulty in determining *amenable* regions of source code. Unlike other works in this field, which have managed to produce results like reduction in runtime, this project did not manage to match or add to these results.

6.3 Limitations and Future Work

The following section will include limitations when implementing this project, and future work that would be carried out if more time was available for the project. This will be split into multiple subsections covering different areas of the projects where there were limitations, and the final subsection will conclude with some recommendations.

6.3.1 The Dataset

The dataset that was used for the project consisted of 8 open-source Java projects. Random Edits were made using the GI tool Gin on methods within the source code, and then metrics were computed over these same projects. Due to time constraints during the project, more open-source projects were not included in the dataset. This is an obvious limitation as it is possible that with more data, a more solid connection or relationship could be found between the feature set and regions of source code that are *amenable*. If there were more time, then this would be a valuable task to carry out as collating many open-source projects and building a larger dataset could mean that more could be investigated, and it is possible that results will be more fruitful and even produces models which could be used to accurately predict *amenable* regions.

6.3.2 Machine Learning Approach

Although this project used the CRISP-DM framework to approach building Machine Learning Models. There were many areas where this approach was not taken advantage of completely. There were originally plans for more iterations to be made where different features would be tested or introduced. There ended up being two main cycles with individual iterations on each of the targets. In retrospect, it would have been more efficient in using this iterative system to have made smaller iterative chunks and then the individual targets could have been investigated more thoroughly. Additionally, unfamiliarity in the area lead to a large portion of time being used up learning about how to use the tools to create the machine learning models. It would have been better if more time at the beginning was spent on this research, combined with research on GI of Software.

6.3.4 Hyperparameter Tuning

In this project, hyperparameter tuning was used to improve the accuracy of the models. However, in retrospect although the approach was systematic, much more time could have been dedicated to Hyperparameter Tuning . The search method used was a random search, and over the four models which were chosen this was ran over 800 iterations. There may have been more improvement in the models if more hyperparameters were chosen to be investigated for each model type. Additionally, the choice of values for each hyperparameter across the search space was kept the same, even in later iterations. It would have made more sense to vary these when looking at the constructed metrics as features to the model. If working on this in the future, it would be much more beneficial to spend more time systematically improving the models, as this could in turn improve the results that were found in this project.

6.3.5 Explainability

In this project, Explainability was used to further explore the relationship between features at the predictions the models make. This was carried out towards the end of the project and due to disruptions and time constraints this was only finished with respect to one of the targets. Although this was enough to demonstrate how it can be useful to give further insight into features it would have been better if this could have accounted for all the features and each target that was defined. If this working on this in the future, this would be finished off so that the full picture can be shown.

6.3.6 Limited Research

The biggest limitation of this project is that there is a limited amount of research in GI of Software. As was stated from the outset of this project, this is a young research field that is still gaining traction and due to that it was difficult to find a larger variety of research to help guide the direction of the project. This project may very well be a better prospect in the future when more breakthroughs are made in discovering *amenable* regions of source code. While this is true, it would have been better to assess even more of the research from the beginning so that a clearer understanding of the mechanisms was gained. If working on this in the future, it would be beneficial to spend more time familiarising with the concepts of GI.

6.3.7 Feature Construction

The feature construction that was implemented in this project was very basic. It was the first to attempt to use the tool JavaParser in this way to create custom Software Metrics for source code. This could have been taken much further and would probably form the basis of a project by itself. The features that were constructed were simply counts of different elements of source code and their positioning relative to other statements, such as being surface or nested "if" statements. It is possible that more complicated metrics, perhaps those which analyse class level statements or are focused on more subtler elements of the source code would produce more fruitful features. It was not possible in the time frame of this project to create such features but given time these would be a worthwhile investigation to discover if more complicated feature construction would have a higher success rate.

6.3.8 Recommendations

If this project was being taken in a future year. These recommendations are likely the most useful to someone picking up the project for the first time:

1. GI of Software is a young research field, so it is possible that in the future there will be more research available. However, it is recommended to have a solid foundation and understanding of the mechanisms of how GI operates. It is very easy to be stuck on this if it is the first time being exposed to the topic.
2. It is recommended that more time be spent having a stronger grasp of the Machine Learning techniques that are available. This would likely mean a much easier step into the overarching goals of this project.
3. A more thought-out design for the Machine Learning implementation at each stage is recommended as this would allow for faster and more efficient progress when getting to more challenging aspects.
4. An understanding of how certain Software Metrics operate is recommended as this is crucial to investigating how they may relate to *amenable* regions of source code.
5. Forward planning of producing and writing drafts is recommended. While this is applicable to all dissertation projects, it would be especially useful for this project as there are a lot of esoteric concepts that may require review to understand.

6.4 Critical Reflection of Project

The following section will be a critical self-reflection of the project as a whole. This will be split into two subsections that cover the areas of strength and the areas that need improvement.

6.4.1 Areas of Strength

In this project, I believe that I was able to produce and investigate a good number of features as well as being able to apply Machine Learning to this problem in a reasonably systematic way. Two main iterations were completed, and I was also able to learn and use JavaParser for feature construction of custom metrics and produce a simple Java program that could compute these metrics over the open-source projects that were provided in the dataset. These were quite simple would have been easy for expansion if I had more time. I believe that I was able to gain a good grasp of the background of GI as well as many of the breakthroughs it has had in recent years, with regards to code repair and other performance increases. The results that I was able to produce show that there is potential for applying Machine Learning solutions to this kind of tough and esoteric problem. I think that the beginning implementation stages of analysing the dataset and understanding how certain features relate to the targets that were set out was probably the strongest part of this project.

6.4.2 Areas of Improvement

In this project, I believe that the biggest drawback was that more time was not spent in the early stages getting a solid foundation and understanding of the Machine Learning tools and techniques I had at my disposal. This led to me being unable to create as many iterations with further investigations and refinements to the dataset. My original plan that was set out in the dissertation outline included completing four main iterations but due to this and other disruptions in the year. My progress was not as complete as it probably would have been otherwise. I believe that lot of my struggling also came down to not being able to communicate the more complex concepts in a project like this in simple terms.

References

[1] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen, “Reversible Debugging Software: Quantify the time and cost saved using reversible debuggers,” 2013.

[2] J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White and J. R. Woodward, "Genetic Improvement of Software: A Comprehensive Survey," in *IEEE Transactions on Evolutionary Computation*, vol. 22, no. 3, pp. 415-432, June 2018, doi: 10.1109/TEVC.2017.2693219.

[3] W. B. Langdon and M. Harman, "Optimizing Existing Software With Genetic Programming," in *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 1, pp. 118-135, Feb. 2015, doi: 10.1109/TEVC.2013.2281544.

[4] David R White. 2017. GI in no time. In Proceedings of the Genetic and Evolutionary Computation Conference Companion. ACM, 1549–1550.

[5] A. E. I. Brownlee, J. Petke, B. Alexander, E. T. Barr, M. Wagner, and D. R. White, “Gin: Genetic improvement research made easy,” in Proceedings of the Genetic and Evolutionary Computation Conference, ser. GECCO ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 985993. [Online]. Available: https://doi.org/10.1145/3321707.3321841

[6] T. J. McCabe, "A Complexity Measure," in *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308-320, Dec. 1976, doi: 10.1109/TSE.1976.233837.

[7] B. A. Nejmeh, “Npath: A measure of execution path complexity and its applications,” Commun. ACM, vol. 31, no. 2, p. 188200, Feb. 1988. [Online]. Available: https://doi.org/10.1145/42372.42379

[8] C. Lee, 1997. [Online]. Available: https://javancss.github.io/#. [Accessed 18 October 2021].

[9] C. Le Goues, M. Dewey-Vogt, S. Forrest and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 3-13, doi: 10.1109/ICSE.2012.6227211.

[10] F. Wu, W. Weimer, M. Harman, Y. Jia, and J. Krinke, “Deep parameter optimisation,” in Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015

[11] B. R. Bruce, J. Petke and M. Harman, "Reducing energy consumption using genetic improvement", *Proceedings of GECCO*, pp. 1327-1334, 2015.

[12] J. Petke et al. 2019. A Survey of Genetic Improvement Search Spaces. In Companion Material Proceedings of the 14th Genetic and Evolutionary Computation Conference (GECCO 2019 companion). ACM, 1715–1721.

[13] N. Harrand, S. Allier, M. Rodriguez-Cancio, M. Monperrus, and B. Baudry, “A journey among java neutral program variants,” Genetic Programming and Evolvable Machines, pp. 531–580, 06 2019.

[14] E. Schulte, Z.P. Fry, E. Fast, W. Weimer, S. Forrest, Software mutational robustness. Genet. Program. Evolvable Mach. **15**(3), 281–312 (2014)

[15] A. Blot and J. Petke, "Empirical Comparison of Search Heuristics for Genetic Improvement of Software," in *IEEE Transactions on Evolutionary Computation*, vol. 25, no. 5, pp. 1001-1011, Oct. 2021, doi: 10.1109/TEVC.2021.3070271.

[16] S. Zuo, A. Blot, and J. Petke,, 2022. Evaluation of genetic improvement tools for improvement of non-functional properties of software | Proceedings of the Genetic and Evolutionary Computation Conference Companion.

[17] M. Smigielska, A. Blot and J. Petke 2021, May. Empirical Analysis of Mutation Operator Selection Strategies for Genetic Improvement. IEEE.

[18] N. Hotz, "What is CRISP DM?," *Data Science Project Management*, 2021. https://www.datascience-pm.com/crisp-dm-2/ (accessed Sep. 01, 2022)

[19] "scikit-learn: machine learning in Python — scikit-learn 0.20.3 documentation," Scikit-learn.org, 2019. [Online]. Available: https://scikit-learn.org/stable/index.html

[20] M. Kuhn, The caret Package. [Online]. Available: https://topepo.github.io/caret/

[21] The R Foundation, "R: What is R?," R-project.org, 2019. [Online]. Available: https://www.r-project.org/about.html

[22] "Eclipse DeepLearning4J," deeplearning4j.konduit.ai. [Online]. Available: https://deeplearning4j.konduit.ai/

[23] "JavaParser - Home," *javaparser.org*. https://javaparser.org/ (accessed Sep. 01, 2022).

[24] N. Smith, D. van Bruggen, and F. Tomassetti, JavaParser: Visited. 2016. [Online]. Available: https://www.livingtech.com.cn/media/javaparservisited.pdf

[25] "checkstyle – Checkstyle 8.22," Sourceforge.io, 2013. [Online]. Available: https://checkstyle.sourceforge.io/

[26] J. Petke, B. Alexander, A. E. I. Brownlee, M. Wagner, and D. R. White, "Program Transformation Landscapes for APM using Gin" (under review with Empirical Software Engineering Journal)

[27] S. Lundberg, "Welcome to the SHAP documentation — SHAP latest documentation," *shap.readthedocs.io*. https://shap.readthedocs.io/en/latest/index.html (accessed Sep. 01, 2022)

Appendix 1

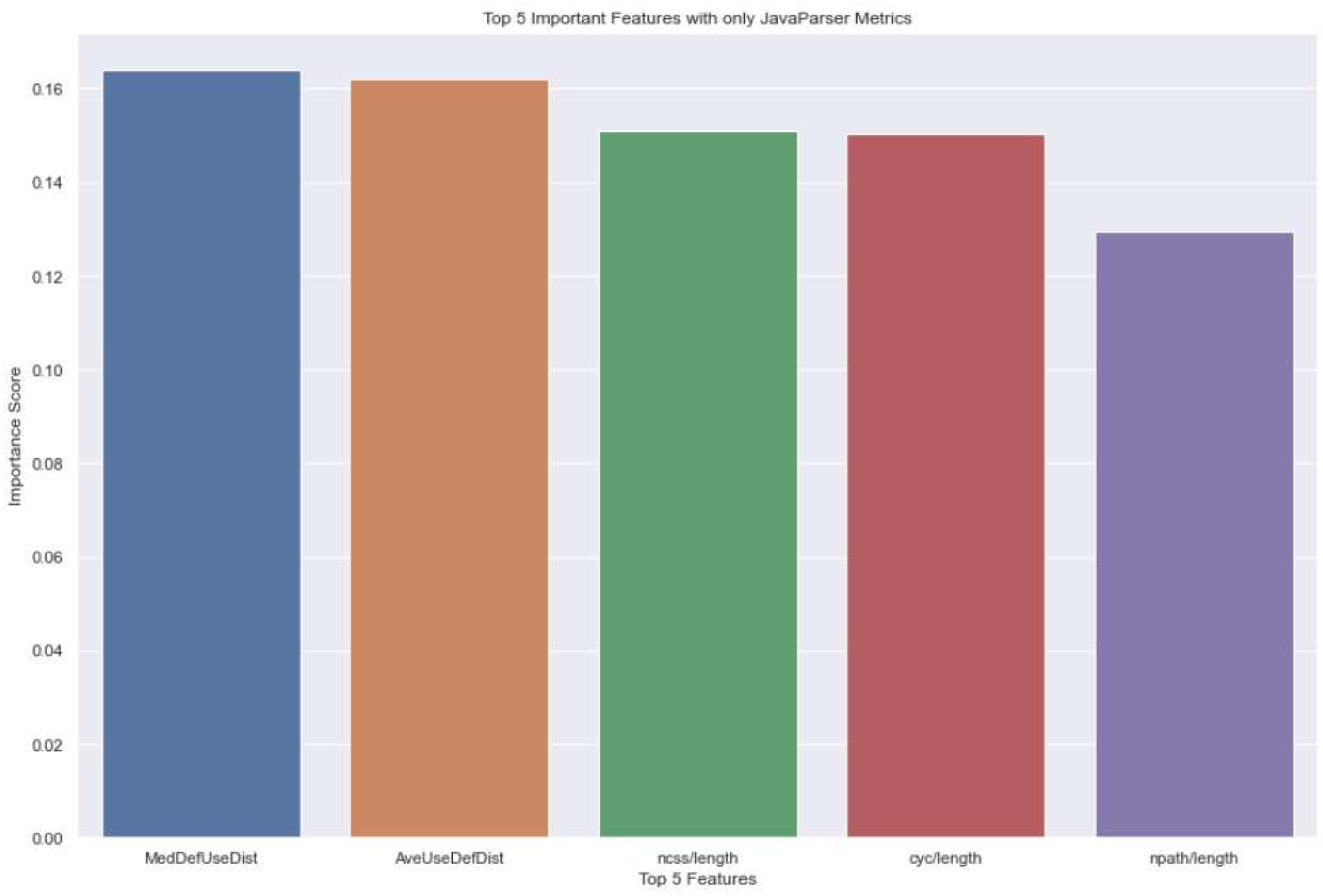


Figure 17. Bar Graph of Top 5 Important features, with only JavaParser Metrics