Guided Metamorphic Transformations for Testing the Robustness of Trained Code2Vec Models

Master's Thesis

Ruben Marang

Guided Metamorphic Transformations for Testing the Robustness of Trained Code2Vec Models

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Ruben Marang
born in Leiden, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Abstract

Machine learning models are increasingly being used within software engineering for their predictions. Research shows that these models' performance is increasing with new research. This thesis focuses on models for method name prediction, for which the goal is to have a model that can accurately predict method names. With this thesis, we could create a tool that can suggest method names to software developers, which would assist in improving the quality of the projects.
This research aims to get insight into the robustness vulnerabilities of a method name prediction model. We use a genetic search algorithm that looks for these robustness problems. The main question this thesis tries to answer is to what extent the performance metrics are affected by applying metamorphic transformations to the test set of a trained code2vec model. Besides this, this thesis also proposes an alternative metric called percentage_MRR, which might better reflect the robustness of a model. The main idea behind this metric is that it penalizes the prediction certainty of a model instead of penalizing the prediction rank.
To answer this research question, a tool is created that runs a genetic algorithm applying these metamorphic transformations to a dataset that a trained model is then evaluating. With this tool, we conducted 22 genetic search experiments on primary metrics and combinations of metrics to see the trade-offs in the Pareto fronts.
The guided search of applying metamorphic transformations on the test set results in an average performance decrease of around 19%. This thesis also compares this drop in performance to the performance decrease a random search algorithm would create. Notably, for every transformer added, the average decrease in performance becomes smaller, and there are transformations, e.g., the if-false-else transformation, that have a bigger effect than others. This thesis concludes that the trained model is not robust against metamorphic transformations and has a significant performance drop.

Title:	Guided Metamorphic Transformations for Testing the Robustness of Trained Code2Vec Models
Author:	Ruben Marang
Student id:	4694511

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. A. Panichella, Faculty EEMCS, TU Delft
University supervisor:	L. Applis, PhD student EEMCS, TU Delft
Committee Member:	Dr. Z. Erkin, Faculty EEMC, TU Delft

Preface

With this thesis, I end my two-year Master's at TU Delft. I completed my Bachelor's degree in Computer Science and Engineering at this university, and this thesis marks the last step of my Master's degree. I have had many opportunities to develop myself and my knowledge here at TU Delft and in my life as a student. This study program has prepared me well for the following chapters of my life.
I chose this project because I was interested in static code analysis. In the different projects I did, in the bachelor's and master's programs courses, I used various code analysis tools before pushing something to GitHub. Then in the master course "Analytics and Machine Learning for Software Engineering" we looked at different possibilities for research. In this research, I came across other papers on method name prediction and thought the tool would be very interesting to look further into. This research into method name prediction has been ongoing for a long time, and I would like to contribute. In combination with this, I contacted Dr. Annibale Panichella, who pointed me to the research of Leonhard Applis on metamorphic testing. His paper on the Lampion project was fascinating, and I combined both research avenues.
Many people helped me with the actual thesis and distractions along the way during this entire thesis. I especially like to thank Leonhard Applis, who was available to me almost daily during the thesis. I would also like to thank Dr. Annibale Panichella, who helped me with his research expertise and was able to point me in the right direction. Both contributed significantly to this research, and I learned much from them.
Lastly, I would like to thank my parents, sister, girlfriend, and everybody else who helped me outside of studying during this period. They ensured that I was not only studying but also still enjoying the other parts of life. I am happy to end this chapter of my life with this thesis and look forward to what comes next.

Ruben Marang
Leiden, the Netherlands
August 2022

Contents

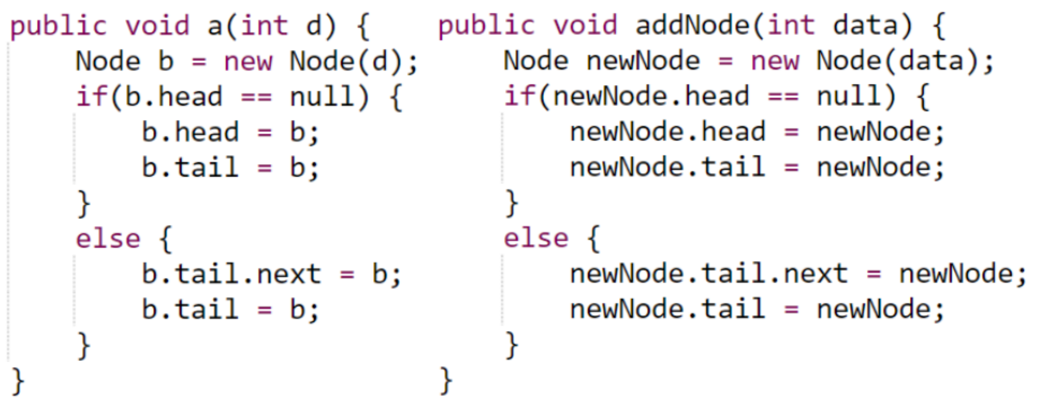
List of Figures

List of Tables

Chapter 1

Introduction

Since writing code has become a more significant part of different industries, more and more problems have come to light. People who might not necessarily be familiar with the best programming practices do have to write proper code that is readable to other programmers. Writing code by itself might be considered easy to learn, but understanding and reading code are much harder. For example, coming up with method names is hard, which can be pretty tricky. So why not help coders by doing it automatically?
Experienced coders will write the code in such a way that one only has to read the identifier names to understand the intended meaning of the class or method. Various studies have been done on the influence of identifier names on code quality, e.g., Butler et al.[6][7]. Here it has been shown that a good naming convention is key to identifying the purpose of the code. An experienced programmer will immediately recognize and understand the function or fragment of code. This fastens the familiarization time of the programmer with the code base and increases efficiency in a company or research group.



```
public void a(int d) {
    Node b = new Node(d);
    if(b.head == null) {
        b.head = b;
        b.tail = b;
    }
    else {
        b.tail.next = b;
        b.tail = b;
    }
}
```

```
public void addNode(int data) {
    Node newNode = new Node(data);
    if(newNode.head == null) {
        newNode.head = newNode;
        newNode.tail = newNode;
    }
    else {
        newNode.tail.next = newNode;
        newNode.tail = newNode;
    }
}
```

Figure 1.1: Differences between unclear (left) and clear (right) identifiers

On the internet, there are countless examples of bad and good code. For instance, the code snippet in Figure 1.1 is an example of clean code on the right and bad code on the left. Both have the same functionality, but it takes longer to understand unclear identifiers than it does when they are clear. Someone who is not familiar with the syntax of Java will still know what the function does just by looking at the names, an experienced programmer will know immediately. Unclear identifier names, however, will make the programmer work in a top-down approach and go over each line of the entire code. For a large codebase, this will quickly become infeasible and tedious. This is just one example, but it is clear that programmers must properly consider the names given to methods and identifiers.

For smaller methods, giving identifier names is considered easier since there are only a few identifiers that the programmer needs to name. However, looking at big classes and methods, it becomes more of an art. Complex classes might have complicated data types, or a method might have a more complex function than just adding a node, as in the example (Figure 1.1). By learning from properly named existing code, underlying patterns may be found.

The problem for the developers occurs when they are new to an already existing project with many classes and methods. When the identifiers in the project are not named properly, it will take the new developers a long time to get familiar with the code base and know what each method does. This is an annoying process, and it costs a lot of time. Developers perform peer reviews to try and fix this, but there are always things that get through, especially in smaller teams. In those circumstances, few people can review merge requests, meaning mistakes like unclear identifier names will get through easier. Programmers often use their experience to perform peer reviews, which makes it hard to find all the mistakes.

Just as programmers might use prior experience to name identifiers, a program can do the same. Based on previous code snippets, a program can learn the names that go with this code. In this thesis, we explore testing a convolutional neural network that can predict method names. A convolutional neural network is a type of machine learning that is often used in classification problems. This will help programmers either check their method names or help them come up with new names for their code. Besides programmers, this thesis will also help data scientists find out how good their model is and hence help developers get better models. The tool in this thesis will evaluate the neural network on data samples with different metamorphic transformations applied to gain insight into how robust the model is.

Other prior works have looked at method name prediction with neural networks and achieved great results. In recent years, code2vec by Alon et al.[3] has been used as a baseline performance for the task of method name prediction. Compton et al.[8] researched the generalizability of these code2vec models (the models created by Alon et al.) by obfuscating identifier names. Their work is discussed in Section 3.2.

These prior works, however, have some shortcomings in terms of robustness. Compton et al.[8] showed that the predictions of the method names are very dependent on other identifier names. This means that unclear variable names will lead to a worse method name prediction. The model is thus not ready for corporate use. Compton et al. tried to fix this by training with different identifier names, which improved the model. This yielded better results than the paper by Alon et al.[3]. Chapter 3 describes these papers and how they relate to each other.

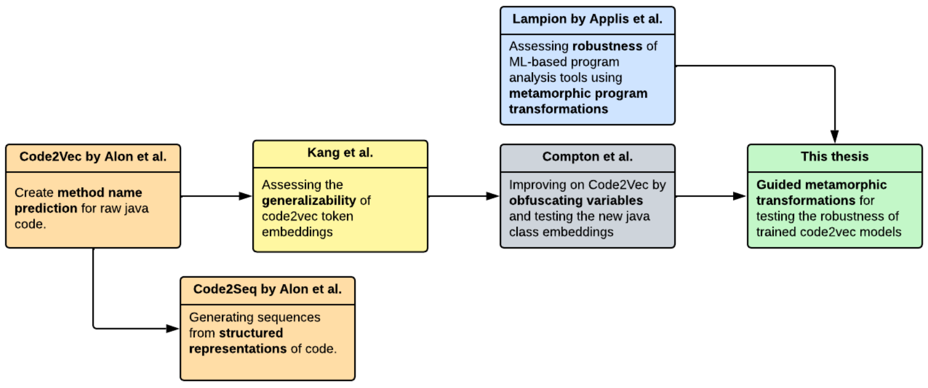

Lampion by Applis et al.
Assessing **robustness** of ML-based program analysis tools using **metamorphic program transformations**
Code2Vec by Alon et al.
Create **method name prediction** for raw java code.
Kang et al.
Assessing the **generalizability** of code2vec token embeddings
Compton et al.
Improving on Code2Vec by **obfuscating variables** and testing the new java class embeddings
This thesis
Guided metamorphic transformations for testing the robustness of trained code2vec models
Code2Seq by Alon et al.
Generating sequences from **structured representations** of code.

Figure 1.2: Relationships between prior works

Researchers can use the tool created in this thesis to evaluate their model against different types of data alteration and, through that, evaluate its robustness. This evaluation can be done according to multiple metrics and used as an extra argument for why a model is better or worse than another.

1.1 Problem statement and research goal

In the previous section, the importance of good method names has been argued. In research over the past years, models have been trained through machine learning to predict these method names. In prior works[19][8], it has been noticed that the models designed for this are not robust to different transformations and rely heavily on the variable names for their predictions. As mentioned, it is a clear problem since, for this type of source code analysis to work in practice, it needs to perform well on unseen data and not have its predictions influenced by bad identifier naming.
This research aims to get better insight into the robustness vulnerabilities of a method name prediction model. This is done by researching the impact of different metamorphic transformations explained in Section 2.5. This thesis will investigate whether using different types of data alteration improves or worsens the performance of a trained model.

1.2 Research questions

When a neural network is tested using an altered dataset, it might generate different results. To know the impact of data alteration, it first needs to be evaluated how the model metrics are impacted. This is why the first research question is:

RQ1: To what extent does adjusting the semantics of the test-set data using metamorphic transformations with genetic algorithms affect the scoring metric?

Next, this thesis will focus on guiding the results that the neural network published by Alon et al.[3]. The results of the test set might improve when the test set is altered using metamorphic transformations. For this, a basic genetic algorithm will be used. This produces different populations and tests which give the best results for a certain metric. The second research question this thesis will answer is:

RQ2: To what extent does adjusting the semantics of the test-set data using metamorphic transformations with genetic algorithms influence the combination of additional metrics?

The current performance robustness metric widely used is mean reciprocal rank (MRR), further explained in Section 2.8. This metric takes into account what rank the correct prediction is but not how certain the model is about the correct prediction. MRR is applied in a wide range of domains where they do not have certainties and probabilities in their predictions. In method name prediction, however, these probabilities are given. This means that there might be a better performance scoring alternative to MRR that can be found. An alternative to this will be proposed and tested on the data in this thesis. Furthermore, the following question will be answered:

RQ3: Is there an alternative performance robustness metric to MRR that can correctly represent the performance of a model?

With these research questions, we hope to gain a better insight into the robustness of the pre-trained code2vec model.

1.3 Outline

The rest of this document is structured as follows: This thesis starts by explaining background information in the next section, Chapter 2. Chapter 2 explains various general techniques and definitions used in the rest of the thesis. The prior knowledge that is expected before reading the document further is stated at the beginning of Chapter 2. In Chapter 3, Related Work, this thesis discusses other papers on related topics that have directly influenced the research approach of this thesis. In Chapter 4 the architecture of the neural network is explained together with the methodology to answer the research questions.
In the section after that, Chapter 5, the experiments are described and how these are set up. Chapter 5 also contains an overview of the packages used. Lastly, Chapter 5 explains what hyperparameters are used and why those are chosen. In the Results section, Chapter 6, the

results of the experiments are described and interpreted. In the conclusion, Chapter 8 we return to the research questions and answer them with our results. Besides the results, this thesis also gives recommendations for future work and the drawbacks of the experiments.

Chapter 2

Background

This chapter provides an overview of the relevant theory for this thesis. Elements of machine learning and experimental computer science will be explained alongside prior work that has already been done in the field. However, some concepts will not be explained, as this thesis assumes basic linear algebra and artificial intelligence knowledge. This thesis also assumes that the reader can read Java source code.

2.1 Convolutional neural network

A convolutional neural network (CNN) is a specific class of artificial neural networks most commonly used to analyze images. They are both shift and space invariant, making them good at classifying and recognizing images. More recently, they have also been used to interpret graph-based source code representations as images. Here the set of values will not be pixel intensities but rather a discrete set of all allowed identifier names[16][42].
A typical neural network consists of a convolutional layer, a pooling layer, and a fully connected layer. In the convolutional layer, the main part of the computational load is done. This layer has a set of learnable parameters called the kernel, usually a 3x3 or 5x5 matrix. The kernel loops over the pixels in the image and performs the dot product each time it moves.
The different layers work together to detect different characteristics in the input image. The first layer might react to a certain orientation of a row of pixels, and then the second layer can detect if the combination of rows is correct. One drawback of these kernels is that they are computationally expensive. That is why most neural networks only have one or two layers with kernels.
In the case of processing an image, it can result in certainty for a label (like some kind of flower or animal), a numerical value (the number of dogs in a picture), or it can be another picture where the input is modified. In the case of this thesis, the input is an image of the graph-based representation of a piece of the source code. The neural network can learn the embeddings in the code and predict a method name for the piece of source code that was given as input.

2.1.1 Supervised vs unsupervised

There are two types of learning within machine learning, supervised and unsupervised. Supervised learning uses labeled data in training. This process aims to create a model that learns the underlying structure from the pre-labeled data. With this underlying structure, the model can correctly predict new unseen data. For unsupervised learning, there is no label for the data. Rather than predicting the data labels, unsupervised learning tries to organize and model the data. Other machines or programs further down the pipeline can use the organized data to do their job more efficiently.
An example of unsupervised learning is data clustering, where the goal is to learn how the data can be grouped into a certain amount of clusters or groups. These clusters can share some characteristics or be based on other criteria. For more software and engineering-related examples of unsupervised machine learning, one can look at the survey done by Khanum et al.[20] that discusses different techniques and methodologies for unsupervised machine learning. Two examples from their paper are clustering and feature selection.
This thesis will use supervised learning where the labels (method names) are known during the model's training. Besides method name prediction, there are also other problems where supervised machine learning is used. An example is the paper by Immaculate et al.[17], where bug prediction is made using a supervised machine learning algorithm.

2.1.2 Overfitting vs Underfitting

When learning new data, the goal is to properly represent all the data, not just the data on which the model is trained. With overfitting, the model mostly memorizes the data from the training set and can make good predictions for this data. However, it will not be able to make good predictions for new and unseen data. When underfitting, the underlying patterns for the data are not found by the model, and it will thus make poor predictions. The goal of the model training is to find a good balance between the two, which results in a generalized model that can predict unseen data. Both underfitting and overfitting can be detected using a validation set. This is a portion of the dataset that the model has never seen. If the loss on the validation set is close to the loss on the test set, there is no under-or overfitting.
A CNN has a full connectivity attribute, meaning that all neurons are connected, making them vulnerable to overfitting and underfitting. Fortunately, both of these instances can be avoided by using cross-validation[33].
Cross-validation uses a different part of the full dataset as the test set for each iteration. To start the full dataset is divided into k segments. In each iteration, one segment is used as a test/validation set, and the rest are used for training. This thesis does this k times, with each time a different segment for the test/validation.
There are also other ways to solve underfitting or overfitting. Some examples of techniques to solve overfitting are data augmentation and regularization. To solve underfitting, the model complexity can be increased, or regularization can be reduced.

2.2 Attention

Attention in deep learning is a mechanism used to represent what parts of the input data the model should focus on. It will be discussed in quite some detail since there have been many developments over the years which are relevant to understanding the context of this thesis.
Attention can be translated into a more relatable example by talking about the human brain. Humans can automatically focus on the main subject of an image and thus ignore the background of the image. The idea behind neural attention is based on mimicking this cognitive attention in the human brain. The original attention mechanism was inspired by this and used in images where the model needed to learn how to focus on the important regions of the image. An example of this is the paper by Xu et al.[41] introducing an attention-based model that automatically learns to describe the content of an image. Bahdanau et al.[5] and later Raffel et al.[31] posed that attention allows for the context to be used to compute the state at each time step. In their paper, they illustrate where the model's attention was in the image and what word corresponded to that region. Their results indicate that attention performs better on the BLEU metrics[28] and METEOR[11]. The thesis will explain the attention mechanism for images since this is easier to understand. The attention layer in the source code application will work the same, except that the input is not a vector representation of an image but rather a vector representation of the source code[3].
There are two steps for a model to determine the appropriate caption, like in the paper by Xu et al. [41]. First, the image is encoded in an internal vector representation v using a CNN. Then, v is decoded into a word vector signifying the captions using a recurrent neural network (RNN).
An RNN, like a long-short term memory (LSTM), not only outputs to the next layer but also has feedback components. This is key to processing a sequence of data instead of only a single data point. Since LSTMs are good for classification tasks, they are used for determining the image captions.
Without attention, the LSTM looks at the entire image to determine every caption word. This is not very efficient since we usually generate different words in the caption by looking at different image regions. To solve this problem, n different, non-overlapping sub-regions such that v_i will be the internal feature representation used to generate the i-th word. When implemented, the decoder will determine a word of the caption by only looking at a specific image sub-region.
The attention determines what image regions are important for a specific word. The weighted arithmetic means of these regions by taking all sub-regions and context as their input and outputs. The arithmetic mean is calculated by taking the inner product of the actual values and their probabilities.

$$E(X) = \sum_n p(X = X_n)X_n$$

These probabilities are determined using the context C. This represents everything the RNN currently has generated as output until that point. The output of the CNN y is applied to weights, which are the learnable parameters for this attention layer. This means that the weight vector is updated as more training data becomes available. An activation function, like the *tanh* function, is applied, so that very high values have small differences between them, and the same goes for very low values. These values will be close to 1 and -1, respectively, and result in the m_i values.

$$m_i = tanh(y_i W_{y_i} + CW_c)$$

These m_i values are then passed on through a softmax function which outputs a probabilities vector s. These probabilities correspond to the relevance of the sub-regions given the context. Given the output of the attention, the LSTM can then determine each word.
This is just one form of attention. Other forms are explained by Vaswani et al. in their paper *Attention is all you need*[39]. They use attention in translation from English to German and English to French.

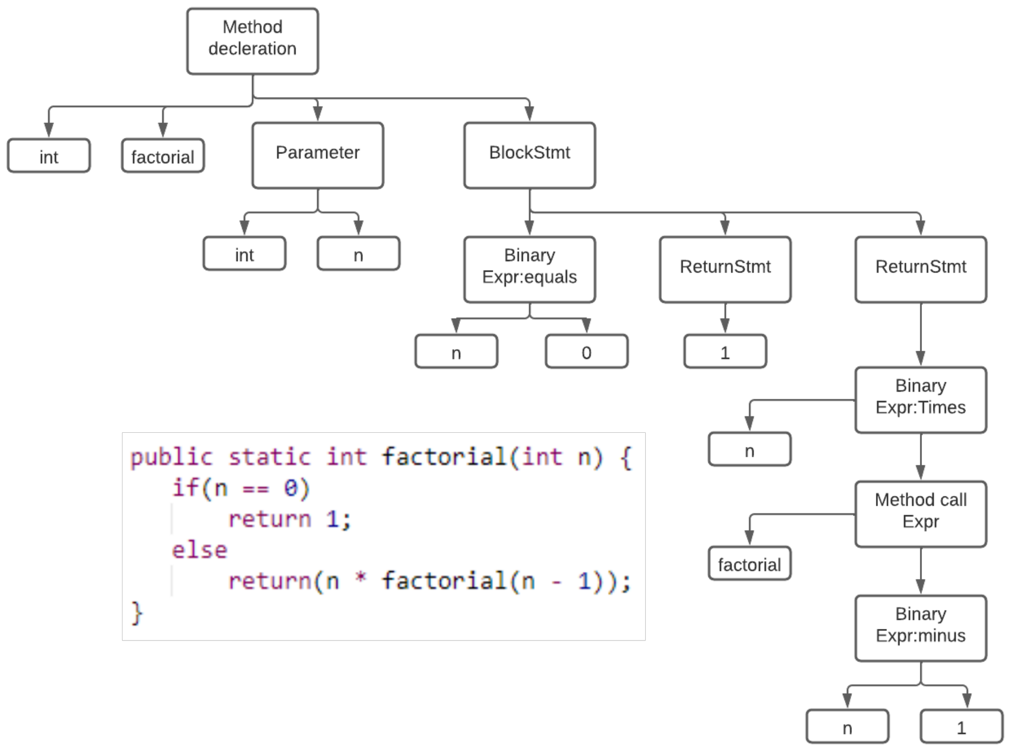
2.3 Abstract syntax tree

An abstract syntax tree (AST) is an undirected knowledge graph that only contains the important structures in the code that make up the functionality. As done in previous works on the same subject, for example, code2vec by Alon et al.[3]. This thesis uses abstract syntax trees (ASTs) as the graph-based representation of the data as input for the model. This knowledge graph is not restricted to a certain programming language but can represent code written in Java, Python, C, and many other programming languages. The Java method factorial and the corresponding AST are depicted within the example in Figure 2.1.

In theory, an AST should be able to use the prediction model across different programming languages since it is not language-specific. The exception is meta-programming, which describes various techniques to alter programs, such as AST manipulation.
In practice, research has also shown that using language-specific features to represent information better pays off, as shown by Compton et al.[8]. In the context of this experiment, the AST tries to deduct the meaning of the program into a knowledge graph and uses this as input for the neural network.

2.4 Method name prediction

Method name prediction predicts a method's name based on the source code. Predicting code properties has been around for a long time, e.g., Raychev et al.[32], and Alon et al. [1], and method name prediction is a part of this. A model outputs multiple predictions with a corresponding likelihood in method name prediction. According to the metric used, the model gets a score based on the predictions. A common metric for measuring the quality of a method name prediction model is the F_1-Score discussed in Section 2.8. Method name prediction is part of automatic source code analysis and can be used in areas like authorship identification or malware detection. Automatic source code analysis has become more and more popular among software engineers. Previous studies have focused on method name prediction as an area of research, e.g. Compton et al.[8] and Alon et al.[3]. These studies showed that the models rely heavily on variable names to perform their predictions. Changing the variable name does not impact the method's functionality, which means it should not impact the name prediction.



```
public static int factorial(int n) {
   if(n == 0)
       return 1;
   else
       return(n * factorial(n - 1));
}
```

Figure 2.1: Code and corresponding AST for factorial method

There are two different embedding approaches for method name prediction discussed within this work, code2vec[3] and code2seq[2]. To enable machine learning to do something with source code, embedding needs to be used. Code2vec does this while maintaining the semantics of the code, which is explained in more detail in Section 3.1. The reproduced paper discussed in the introduction[8] uses code2vec while code2seq is considered an improvement over code2vec. Code2seq keeps word order intact instead of bundling the words up at the end of the AST. Compton et al. noticed that experimenting with code2vec relies heavily on identifier names for the method name prediction. Code2vec itself converts the Java code into an AST and learns to combine the set of paths used to construct the AST using an attention mechanism.

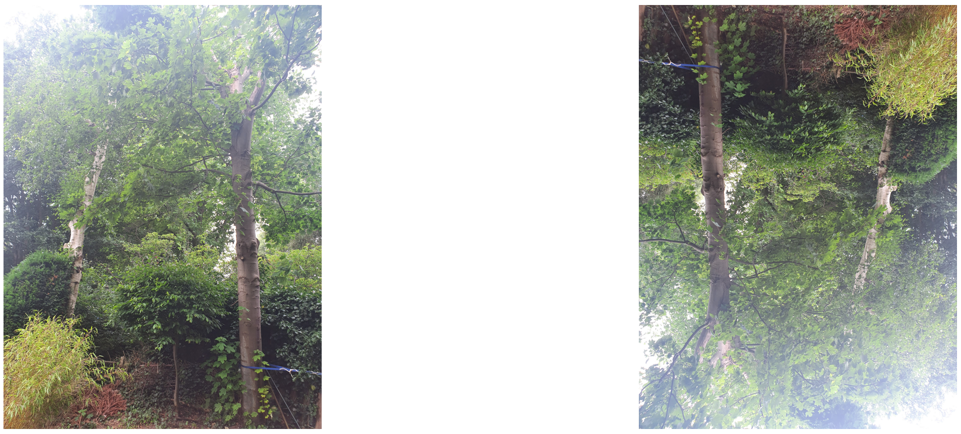
2.5 Metamorphic testing

Metamorphic comes from a rock that has undergone a transformation and now has a different look. A metamorphic relation (MR) refers to two functions with different inputs but the same output. An example of this can be $sin(x)$ and $sin(x+2\pi)$, which are different formulas but result in the same output. These metamorphic relations are used during metamorphic testing.

Metamorphic testing is not only being used in software engineering but also in other fields, such as image classification. First metamorphic testing and its use will be discussed in general and then more specific for the software engineering field.

2.5.1 General

Zhou et al.[43] made a meta survey where they discussed in detail what metamorphic testing is and its applications in different fields. This includes metamorphic testing applications in graph theory, computer graphics, and compilers. Metamorphic transformations (MT) became widely known and popular in machine learning by improving the models for image classification. The transformation creates a different image from the old one with a slight modification. An image with a tree still needs to be classified as a tree if it is moved to the right or turned upside down. These slight modifications need to be recognized by the model and still classified correctly. The aforementioned example is given below in Figure 2.2. Here we can see the same tree first with the normal X-Axis and then with a flipped X-Axis. A classifier must be able to classify both as a tree even if the tree is upside down.



(a) Tree with a normal X-Axis

(b) Tree with a flipped X-Axis

Figure 2.2: A tree with the X-Axis flipped

Adding transformed images like these to the training dataset tunes the weights and makes the model more robust to transformations. Examples of metamorphic testing used outside of source code analysis include Mekala et al. concerning their work towards adversarial detection and metamorphic testing[23]. They were the first to attempt using metamorphic testing in deep learning models built using transfer learning. Transfer learning is a way of learning where the trained model for one task is used as the starting model for another task. In other words, this is an approach to learning with pre-trained models instead of starting from scratch.
Another example of metamorphic testing is the work of Dwarakanath et al.[12]. They created a tool that automatically creates test cases for machine learning applications based on metamorphic testing. This can be very useful when a new application is just launched to see whether it is robust against metamorphic transformations. If one of the metamorphic rela-

tions does not hold, the program will see this as an implementation bug. Besides these two examples, many other papers have also successfully used metamorphic testing in machine learning like Murphy et al.[26], and Xie et al.[40]. The success of metamorphic testing has caused it to be adopted in source code analysis and software engineering.

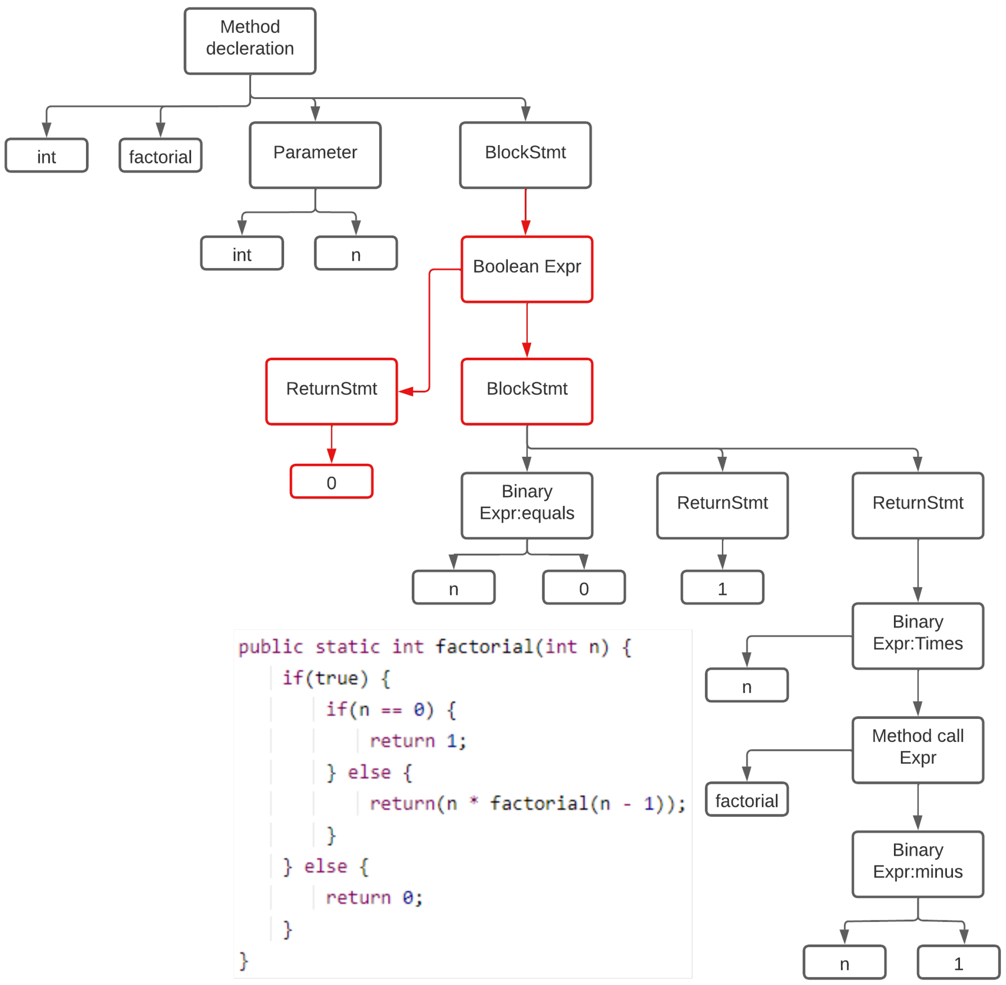
2.5.2 Software engineering

In the work by Compton et al., they used these same transformations on source code to make the method name predictions more accurate[8]. The transformations do not change how the code works but do change the underlying structure of the code. This means that the AST changes, but it should not impact the model's prediction since the functionality is still the same. Consider, for example, when an if-true-statement is put around any code block, it would not change the functionality, but the size of the AST increases. Since the neural network's predictions are based on the AST, these transformations might negatively impact the model's performance. This, however, should not be the case, and the prediction should remain the same. When the model has not been trained on these metamorphic transformations, a slight change in the AST may also cause the predicted label to change.
Below the figure of the factorial method, Figure 2.1, is adjusted with an extra if-true-statement and its corresponding AST. In Figure 2.3, the changes in the AST are highlighted. This corresponds to the *if(true)* statement in the source code. The rest of the AST is the same, but a slight change like this can be enough for the model to predict a different label.

If the model is trained on a wider variety of transformations, it should improve the model when run on the test set. This is expected because the model will learn from these transformations and will learn to predict the methods with an MT correctly. Besides the MT mentioned above, other examples of these transformations are: adding a random unused variable, introducing an unused variable, or renaming variables. All of these transformations have different impacts on the performance of the model. In the paper by Applis et al., these different MTs were tested, and their impact is discussed in Section 3.3 of this thesis. The paper by Compton et al.[8] discusses how MTs improve the robustness of a model and will help with the performance.

2.6 Selection methods

Working with all methods in the test set within the experiments has shown to not always accurately represent the quality of the models[8]. So Compton et al. looked into filtering the test set data to evaluate model performance and eliminate predictions that are not useful. For this thesis, using the entire dataset was not feasible due to computation and time constraints. Because of this, the impact of different selection methods is discussed and put into context with other related work.
Selection methods are ways of filtering the data in the test set. This was inspired by the idea that certain features in feature selection are irrelevant and can negatively impact the performance of the model[10]. Examples of selection methods are *all*, *top-K*, and *random-K*.
Selection methods take a different subset of source code methods into account for testing.



```
public static int factorial(int n) {
    if(true) {
        if(n == 0) {
            return 1;
        } else {
            return(n * factorial(n - 1));
        }
    } else {
        return 0;
    }
}
```

Figure 2.3: Code and corresponding AST for the changed factorial method

Because of this, they will also yield different results. The *all* selection method evaluates the model's performance based on all methods in the file. *Top-K* only select the longest methods in the file. This is based on the assumption that longer methods contain more information about the file and will thus give a better prediction. Lastly, *random-K* selects k random methods for the evaluation.
These selection methods have been used in different prior works to gain different insights into the performance of the models. This thesis will try to represent the entire test set we use to evaluate the model, which is done by taking the *random-K* selection method.

2.7 Genetic search

Genetic search is a technique based on biological evolutionary theory. It explains how species can evolve over the years to better adapt to the environment, thus having a better chance of survival. The phenotype is how a species looks, and the genotype is the DNA inside a living organism.
Genetic search is considered a technique for evolving programs instead of species. Start from a random population of genotypes and fit it to a specific task by evaluating each population and seeing if a small mutation will positively affect the outcome. For every program, these genotypes will be different. This desired outcome is always the maximization of a certain fitness. However, not all fitness functions look for the biggest value but always look for the optimal value. Three different operations are used to achieve this: selection, crossover, and mutation. The selection operation selects the parent individuals with the best fitness from the population to create the individuals for the next population. The crossover operation involves swapping random parts of the selected parents to create the individuals for the next generation offspring. Lastly, the mutation is substituting a random part of an individual with some other random part. Both mutation and crossover are usually done with a probability for each individual. For example, the mutation probability determines how many chromosomes should be mutated in one generation.
A way to explain genetic search is the hello world example. The program tries to create a string that states *”Hello World!”* with all characters and numbers as options. For this example, the initial population would be a string of random characters and numbers of length 12. One individual in this initial population could be something like *j()*=hh15L!k*.
In each iteration, a new population is created where each individual represents a random string as offspring of its parents, except for the initial population, which is fully random. After each individual’s fitness is calculated, the string that is most like the string ”Hello World!” is given the best fitness. With tournament selection, two random individuals will be chosen as the next generation’s parents. Tournament selection is selecting a given number of individuals and then choosing the best two parents out of that set of individuals to be the next generation’s parents. As mentioned above, crossover and mutation are applied with a certain probability on the parents to create the whole population. In the end, the result will state ”Hello World!” and have a fitness of 0 when the problem is seen as a minimization problem. For further reading, the paper by Sette et al. on the basic concepts of genetic search[34] is recommended. For more detailed explanations and examples, this is a great paper to review.
Genetic search is also used in more complicated problems. Forrest et al.[15] came up with a genetic search approach to automated software repair. They define the fitness function using test cases to see how well the program repairs the program bug while still doing everything else correctly. They use an oracle, a machine that knows the answer, to determine whether the program’s output is correct.
Using genetic search with complex problems is often paired with an expensive fitness function. When the program takes more time to calculate an individual’s fitness, it is common to practice having a smaller population size. While in the *Hello World!* example, calculating the fitness function is very inexpensive, and we thus can have a larger population size. A

larger population size means more diversity in the population and often a quicker, stable solution.

2.8 Metrics

During the model's training, the parameters are calibrated so that the model can make predictions that match the expectations as closely as possible. Since supervised learning is used, the expected labels are predefined in the dataset. The weights of the model are configured based on the training set. When a prediction is made during training, the model will evaluate this prediction compared to the true/expected label. The model will tune the parameters based on how wrong the prediction is.
This training data, however, is not what is used to measure the quality of the model. For this, fresh data that the model has not seen is used. The model is given a score based on different metrics on this data. The trained models' scores are compared, determining which model is the best. In this section, the two methods of scoring that are relevant for this thesis are discussed.

F_1-score

The F_1-score is a common measure often used to evaluate overall model performance and is defined as the harmonic mean of precision and recall. To calculate the precision, the portion of true positives is measured and compared to all results that were predicted positive[18]. Recall is the fraction of true positives divided by all elements that were expected to be positive (false negatives + true positives), which is all false negatives and true positives together. The formula for this is written out below.

$$\frac{\textit{true positives}}{\textit{true labeled elements}}$$

The F_1-score determines if there is a good balance between the precision and recall for the prediction model. The highest possible value for the score is 1.0, which indicates perfect precision and recall.
The problem of method name prediction is a little bit more complicated. In this prediction problem, a set of sub-predictions need to be evaluated. This means that there is a set of expected sub-tokens and a set of predicted sub-tokens of which the precision and recall are calculated.
As an example, the expected method name *getName* is taken and the predicted method name as *getSubscriberName*. When this is translated to the expected and predicted sets of tokens the following sets are created: $\{get, name\}$ and $\{get, subscriber, name\}$ respectively. Since both the tokens *get* and *name* are correctly predicted a precision value of 0.67 is achieved. When calculating the recall, it can be observed that both tokens that were expected are also in the predicted set. This means that this value is 1. The F_1-score is calculated next by using both these values, which results in an F_1-score of 0.8. This gives a better picture of

the prediction than just right or wrong. The formula used for this final calculation is:

$$F_1 = 2 * \frac{precision * recall}{precision + recall}$$

Mean reciprocal rank

Mean reciprocal rank (MRR) is a statistical measure for ranking the possible predictions of the model. The responses are ranked based on the likelihood percentage that the model gives a specific prediction. MRR considers which rank is given to the correct prediction and penalizes the model accordingly.
For example, if the model generates a list of [monkey, gorilla, cat, dog], where *monkey* is ranked first, and the correct answer is cat, then the score given is $\frac{1}{3}$. This score is given because *cat* is ranked third in this set. When in the same example, the correct answer is monkey, then the MRR score would be 1.
Given that multiple methods are being evaluated, multiple sets of possible answers with corresponding ranks are calculated. When each method's rank scores are calculated, the mean can be taken to get the MRR over the whole test set.

Chapter 3

Related Work

This chapter discusses related prior works that have contributed to this thesis. The information presented here will be used to conduct the approach and experiments used in this thesis.

3.1 Code2vec: Learning distributed representations of code

The papers by Alon et al.[3][1] have already been mentioned several times in this thesis. They showed how the syntactic paths between identifiers in a program's AST could be useful in predicting names for identifiers[1]. The results of this paper show that these path-based features increase the prediction accuracy.
In the paper *Code2vec*[3] they subsequently showed how these path representations could be used to continuously embed discrete code fragments into a finite-dimensional real vector space. These fragments serve as input features for a deep neural network. The embedding techniques in this paper were heavily inspired by prior work by Mikolov et al.[25][24] about the representation of words into real vector space. The main addition to this method was the addition of an attention mechanism that allows the model to learn the important part of the context.
The results show that their code embeddings perform well at preserving semantic relationships between code fragments. Besides, the models do well at predicting the method names in files across different projects. The scores are depicted as the F_1-Score, which was explained in Section 2.8. They chose this method instead of, for example, the BLUE score[28] because it does not favor short method names that are uninformative. This makes sure that the predicted method names are of good enough quality.
One big drawback of their model is that they are heavily dependent on variable names because of the way they train. Their model training is done on open-source projects with a good variable naming convention.
Previous studies have pointed to weaknesses in the code2vec model, as will be described in more detail below. This thesis will add to this literature by evaluating the impact of multiple metamorphic transformations on the trained code2vec model.

3.2 Embedding Java classes with code2vec

The work by Compton et al.[8] builds on prior works on method name predictions through a convolutional neural network, for which they evaluate *Code2vec*[3]. *Code2vec* converts java code into a vector representation that can then be used in a neural network. To do this, it first needs to convert the code into a set of paths that forms the code's abstract syntax tree (AST). This AST contains identifier names, on which the original *Code2vec* model heavily relies for its predictions. This dependency is a problem since it causes the model not to be generalizable to unseen java code. As mentioned in Chapter 1, having a good naming convention in larger codebases is quite valuable. Having a model that can check the method names in the pipeline of a project will reduce the number of non-descriptive identifier names and thereby improve the quality of the code. Assessing code quality is usually a step in this pipeline. However, most programmers know this is constantly being expanded by having more rules and regulations to improve the quality of the entire code base.
To formulate their research questions, Compton et al. look at the paper by Kang et al. on the evaluation of *Code2vec* embeddings[19]. Kang et al. found that discarding the embeddings representing the variable names increases the performance of their trained model. Compton et al. continue this research and attempt to answer whether the obfuscation of variable names yields an improvement and how they should aggregate embeddings for methods to accurately describe the entire class.
To answer the research questions, Compton et al. used seven different smaller datasets, all mined from open-source projects, which are used to test the accuracy of the embeddings that were learned. In this representation, all datasets had never been used before in machine learning literature. In addition to these datasets, four different embedding models are used, two obfuscation methods, a standard baseline (starting point), and a reduced embedding trained on a reduced dataset. The embedding models are ways that the datasets are changed to train a different model.
The obfuscation methods are ways to change the variable names. The methods used are type obfuscation and random obfuscation. With type obfuscation, the variable names are changed to include the type of the variable, e.g., an integer field might get renamed to $local\_int\_1$. Random obfuscation grabs random characters and creates a variable name from this. Through both obfuscation methods, the real variable names are hidden, which causes the model to be less reliant on these variable names for its predictions. Besides the two obfuscation methods, the standard baseline dataset is slightly bigger than the obfuscated datasets. Because of this, they chose to include a reduced dataset with the same number of examples as the obfuscated datasets.
During experimentation, the datasets were used to train the *Code2vec* models. Using the different selection and aggregation methods. The selection methods used were *All*, *Top K* and *Random K* where $K = 5,3,2,1$. The base aggregation functions used were *max, min, sum, mean, median* and *standard deviation*. All combinations of the powerset up to size two were tested for the aggregation functions.
From this experimental setup, they concluded that the performance of *Code2vec* reduces when the variable names are obfuscated concerning the method name prediction task. Variable obfuscation is the obfuscation of variable names through type or random obfuscation.

However, the embedding evaluation showed that the code semantics are better preserved. Through these results, they conclude that variable obfuscation is a way to increase the robustness of machine learning models on source code. This decreases the bias towards variable names for the particular task of method name prediction.
Besides evaluating the impact of obfuscating variable names, as done by Compton et al., this thesis also evaluates the effect of multiple metamorphic transformations on the performance of the code2vec model.

3.3 Lampion

A relevant paper is the paper Lampion[4] written by Applis et al. since they researched the robustness of machine learning models using metamorphic transformations, which have been explained in Section 2.5. The project's goal was to see what a model understands, and code summarization could give a clearer insight into this than cold metrics. The main idea behind the project is to see whether the model's performance is affected when metamorphic transformations (MTs) are used on code. They also investigate which MT has the biggest impact on the model's performance. The approach presented can be used for testing the robustness of models towards MTs on the source code. When the model where the MT has used scores significantly better, it means that the model has a potential weakness towards that specific metamorphic relation.
To test this, CodeBERT[13] was used, particularly its downstream task of code summarization. Code summarization is a task that tries to understand the code and automatically generate a description for this code. Their metamorphic transformer works at the source-code level for Java programs. Examples of the transformations used are: renaming a class, method, or variable, wrapping a random expression in an if (true) statement, or adding unused variables. All of these transformations change the source code but not the functionality of the source code. However, they do impact the abstract syntax tree (AST) that is being created, which could influence the score in the used performance metric. They also need a model and an existing benchmark of .java files or sufficient pre-and post-processing to transform the data points.
They apply the MTs to all data points in the test set in their experiments. This results in different code samples, which are run through the performance metrics alongside the original code samples. In these samples, the BLUE4-Score[28] is used as a performance metric, which is standard for code-to-text and text-to-code generation tasks. Here the text is generated into n-grams, where n is the number of characters taken for each element of the set, and then compared between the two samples. The Jaccard distance[21] was calculated next to measure the percentage difference between the two different Java-doc comments that CodeBert generated, which measures the difference between the models. The Jaccard distance calculates the difference between two sets. The formula for this is written out below. Here $\cup$ stands for the union of the two sets A and B, and $\cap$ stands for the intersection.

$$JD(A,B) = 1 - \frac{A \cap B}{A \cup B}$$

An example of Jaccard distance in natural language processing (NLP) is the work by Singh et al.[35]. They measure the similarity between two of the same news items on different sites and in different languages.
To establish significance, they used statistical tests on the achieved results. This test concludes that adding random unused variables impacts the BLUE4-Score the most, and MTs have a statistically significant impact on the BLUE4-Score.
This thesis uses the same metamorphic transformations as Applis et al. but evaluates the impact of these on method name prediction.

3.4 Generalizability of neural program models

Another relevant paper by Rabin et al.[30] discusses the generalizability of three state-of-the-art neural network models. This paper points out that many neural programs with public repositories are tested on various datasets and perform well. However, how they perform on unforeseen source code is widely unknown, meaning that the generalizability of the models is also widely unknown. To test this generalizability, the paper uses a generalizable neural program model that should perform equally well on code with the same functionalities. In other words, they use semantic-preserving transformations on their data to test the model's generalizability.
As mentioned earlier in this thesis, the generalizability of a model is crucial if it is to be implemented on a larger scale. The models should be able to perform well on unseen data. Thus knowing how well a model is generalizable is key to seeing how it can be implemented in other situations.
To achieve this, they compared the results of different models for the method name prediction task before adding the semantic-preserving transformations. For this, three Java datasets and three neural network models for code are used, *code2vec*[3], *code2seq*[2], and *GGNN*[14]. These result in nine models which are used for evaluation.
Rabin et al. s experiments show that even with a small change to the programs, these models often fail to generalize their performance. The findings conclude that models based on data and control dependencies in programs generalize better than models based only on ASTs.
These results were gathered for one task in Java programs for the models *code2vec*, *code2seq* and *GGNN*. Whereas this paper has highlighted general weaknesses in the code2vec model, this thesis will compare the impact that different semantic-preserving transformations have on the performance.

3.5 Code2seq: Generating sequences from structured representations of code

Code2seq is a code-to-sequence model focused on the ability to generate natural language sequences from source code, e.g., summarization. The *Code2Seq* they presented is an alternative approach to sequence-to-sequence models that uses the syntactic structure of programming languages[2].
It is similar to *Code2vec* but performs significantly better at the code summarization task,

as presented in the paper. They also measured the performance using the F_1-Score compared to the length of the input code. Across all method lengths *Code2seq* performs better. One of the main differences is that the models of *Code2vec* can generate unseen sequences compared to *Code2vec*, which has a closed vocabulary.
Because of the better performance of *Code2seq* for the code summarization task, this thesis will also incorporate these models and try to replicate this performance on the method name prediction task. We also keep in mind that Rabin et al.[29] evaluated the generalizability of both *Code2vec* and *Code2seq*. One observation he made was that both are vulnerable to different transformations. Rabin et al. also observed a direct correlation between the size of the methods and their sensitivity to changes in the prediction under transformation.

Chapter 4

Approach

This chapter gives an overview of the technologies used. This chapter assumes basic knowledge of genetic algorithms and neural networks. These subjects are introduced in Section 2.1 and Section 2.7 for neural networks and genetic search, respectively. Section 2.2 introduces neural network attention, which is used in the code2vec models and discussed further in Section 4.4 regarding the architecture of the code2vec neural network.
In Figure 4.1 the standard code2vec (dark blue) and the additional operations for the search (light blue) are depicted. These added components showcase where our tool is added to the standard code2vec. The Lampion project by Applis et al.[4] is also part of this diagram since this thesis uses transformers created by them.

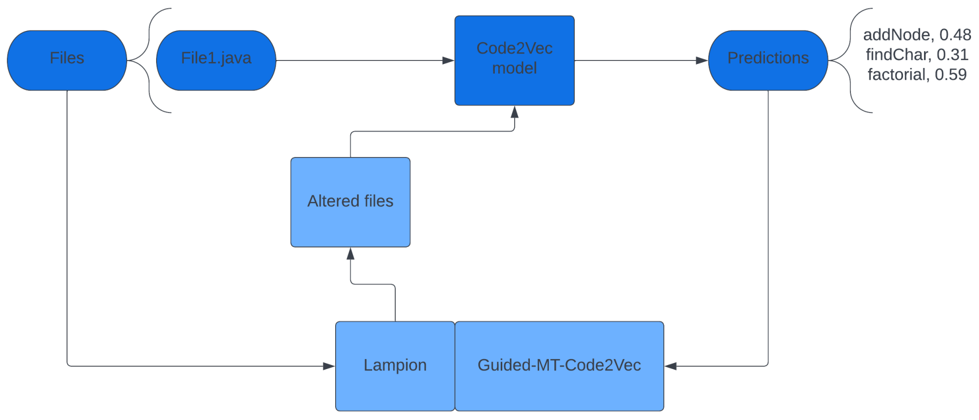


Figure 4.1: Additional operations for searched code2vec

Before the program runs, we first need to define the user requirements that specify the metrics based on which we will calculate the fitness. This is then implemented in the configuration file, which can be changed to optimize based on different metrics or a combination of different metrics.
The program's first step is to evaluate the performance of the initial dataset without any metamorphic transformations applied, which is used as the base value or null hypothesis,

which is further explained in Section 5.5. After that, the genetic algorithm, as explained in Section 2.7, is used to determine the best combination of transformers for the chosen metrics. To do this, the transformations are randomly applied to the dataset, after which the genetic algorithm determines which combination of transformations is the best for the current configuration. All metamorphic transformers used in this thesis are explained in Section 4.2.
Within the genetic algorithm, an individual's fitness is calculated by running the transformations on the dataset and then running the dataset through the code2vec model. The outputs of the code2vec model are predictions with their respective certainty. These predictions can then be parsed by different metrics given by the configuration file to calculate the score. Depending on the configuration, all the metrics combined with their respective weight can then determine the individual's fitness.

The pipeline of our program is depicted in Figure 4.2. First, the metrics and transformer engine properties are initialized. Then the program creates the initial population for the genetic algorithm, after which we evaluate its fitness by applying the transformers and then evaluating the dataset with the code2vec model. Afterward, we calculate the fitness of each individual in the population and evolve the population. The program ends when either the amount of generations has been steady for too long or if the maximum time has passed.

4.1 Generalization and key metrics

This thesis discussed the generalization problems with source code models in Section 3.4. Another way of describing generalization is how well a particular model performs on data it has not seen before. During the training of the neural network, there needs to be enough variance in the data so that the network can adapt to a wide variety of data points. This should make the model more robust to different kinds of data during testing. One way the degree of generalizability is measured is metamorphic testing.
The program of this thesis uses the metamorphic transformations from the Lampion project and therefore uses the transformers built by the Lampion project. These transformers are discussed further in Section 4.2.
The work by Compton et al., discussed in Section 3.2, has already proven that the robustness of trained code2vec models is vulnerable to variable obfuscation. In this thesis, a better insight will be gained into the influence of the other metamorphic transformations. In most papers on method name prediction, the MRR and F1-score are used as key metrics to assess the performance of the trained code2vec models. This thesis will also look into other metrics and can maximize or minimize a combination of these metrics. This will give us another insight into the model's performance and the influence of metamorphic transformations on the performance. If the metamorphic transformations greatly influence the metrics, then the trained code2vec model does not generalize well with respect to that particular combination of metrics or single metric. This is discussed in more detail in Section 4.5.

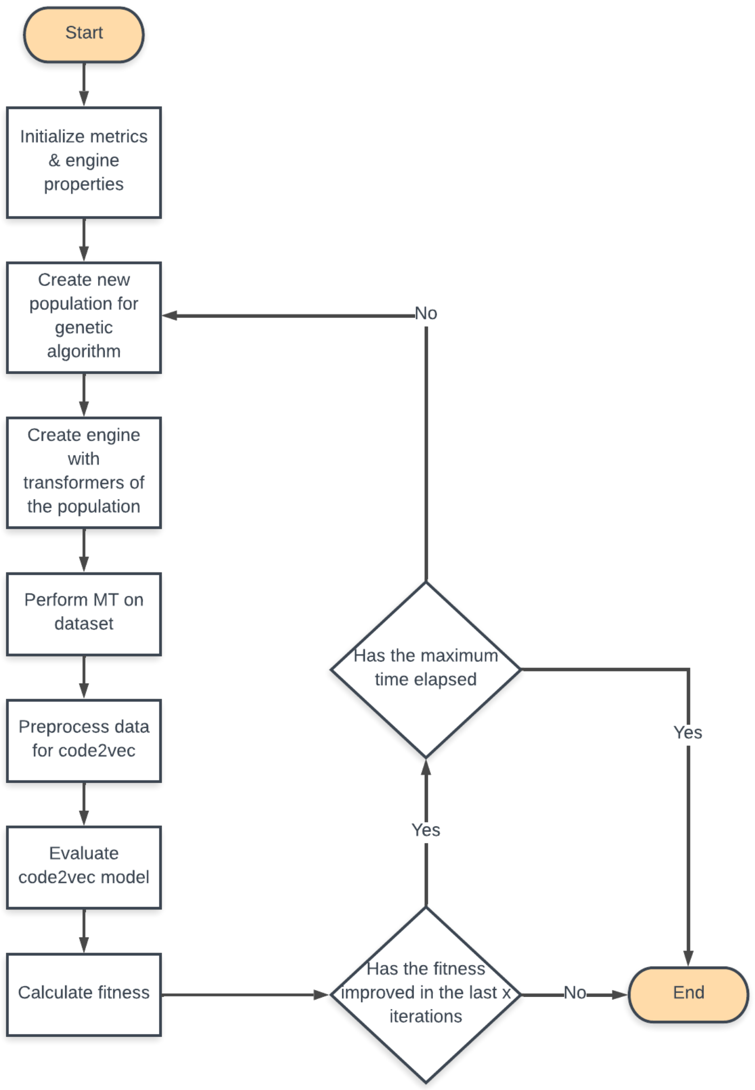

Start
Initialize metrics & engine properties
Create new population for genetic algorithm
Create engine with transformers of the population
Perform MT on dataset
Preprocess data for code2vec
Evaluate code2vec model
Calculate fitness
Has the fitness improved in the last x iterations
Yes
No
Has the maximum time elapsed
No
Yes
End

Figure 4.2: Processing pipeline

4.2 Transformers

In this thesis, several transformers are used that may influence the predictions of the trained code2vec model. These transformers are explained below with examples. The paper by Applis et al.[4] gives a more detailed explanation of these transformers. The Lampion Github[1] also explains the transformers well. All transformers and their respective explanations are listed at the bottom of this section in Table 4.1.

[1]https://github.com/ciselab/Lampion

These transformers are chosen at random during initialization and mutation. Its transformers are applied to the dataset to calculate an individual's fitness. Each transformer implements a different MT. For instance, the IfTrueTransformer puts an if-true statement around a code block. This means that the code will always get executed and only changes the underlying AST, not the functionality. If there is a return statement in the method we are transforming, the transformer will add an else statement and return a neutral element. Another example is adding a neutral element. If we add a neutral element to an integer, double or long would add 0 to the variable. Adding it to a String would be adding an empty string to it.
A visual representation of all transformers can be found in Appendix A.1.

Transformer	**Explanation**
IfTrueTransformer	Add an if-true statement to a method
IfFalseElseTransformer	Add an if-false-else statement to a method
RenameVariableTransformer	Rename existing variable
AddNeutralElementTransformer	Add a neutral element to a statement
AddUnusedVariableTransformer	Add an unused variable
LambdaIdentityTransformer	Add a lambda identity to a method call
RandomParameterNameTransformer	Rename a parameter

Table 4.1: Transformers and their explanation

4.3 Genetic algorithm

The general purpose of a genetic algorithm has already been explained in this thesis, and the theory behind genetic search has been explained in Section 2.7. This section will focus on this thesis's specific implementation of its genetic algorithm. For this, we did not use a library since these libraries did not easily support variable length for the genotypes. In our program, we need to mutate the individuals and increase the number of transformers we use in every generation. We also want the opportunity to use transformers twice or even three times, which many other frameworks do not support.
As mentioned at the start of this chapter, the genetic algorithm is used to determine which combination of transformations gives the best performance for a certain metric or multiple metrics. Some of these metrics have already been explained in Section 2.8 and the others are explained in Section 4.5.

4.3.1 Genetic algorithm: Implementation details for MTs

In Table 4.2 there are several definitions that are used across this subsection. The chromosome is defined as the genotype of an individual, in the case of our program, the list of transformers, as depicted in Figure 4.3. The initial population is created by creating the metamorphic individuals. A metamorphic individual consists of a list of metamorphic transformers that are applied to the base dataset. The metamorphic individuals are initialized with either 1 or 2 random transformers for the initial population. All individuals in this initial population are evaluated, and their fitness is cached. This is done because the fitness calculation is quite expensive.

Genetic algorithm concepts	**Definition**
Chromosome	List of transformers
Phenotype	Applied transformers to a datapoint
Population	A set of chromosomes together at time t
Metric	A way to measure the performance of a model
Fitness	The metrics applied to the chromosome used to guide the search

Table 4.2: Genetic algorithm concepts and their definition

The fitness calculation consists of first running the MTs on the original dataset. This creates an output dataset on which we run the preprocess script so the trained code2vec model can then evaluate it. This is necessary because the neural network can only evaluate a dataset that was first preprocessed by that script. The model then returns results that are parsed into the metrics of interest for the fitness calculation. These metrics, together with their weights, which are configured according to the hyperparameters of the experiments, calculate the fitness of this metamorphic individual. If the population size is five, this process must be performed five times per generation. This can become very expensive, so efficiency is key.
After a population is evaluated, it is evolved to the next generation through tournament selection, crossover, and mutation. Tournament selection takes one parameter, the tournament size, which indicates how many random individuals the program takes for the selection process. If the tournament size is set to 4, the algorithm will take four random individuals and take the best one from that. Our genetic algorithm does this tournament selection twice to create the two parents used for crossover.
In the crossover process, the two parents produce two children individuals. The algorithm uses the crossover rate to determine which child gets which gene. If the randomly generated number is smaller than this crossover rate, then the first child gets the gene from the first parent, and the second child gets the gene from the second parent. If the randomly generated number is larger, the opposite happens. The crossover process is depicted in Figure 4.5. This is done for the entire population, and afterward, the mutation is randomly introduced. The mutation consists of randomly increasing and decreasing the size of the individuals. The individuals' size increases when the randomly generated number is smaller than the predefined rate. Otherwise, the individual's size is decreased. Both these operations are depicted in Figure 4.4. The increasing rate will always be higher than 0.5 so that the further we get in our evaluation process, the individuals in our population get a larger list of transformers as their genotype. After the population is evolved, the whole process starts over, and all individuals are evaluated against the trained code2vec model.

The program terminates when the best fitness has remained steady for a certain amount

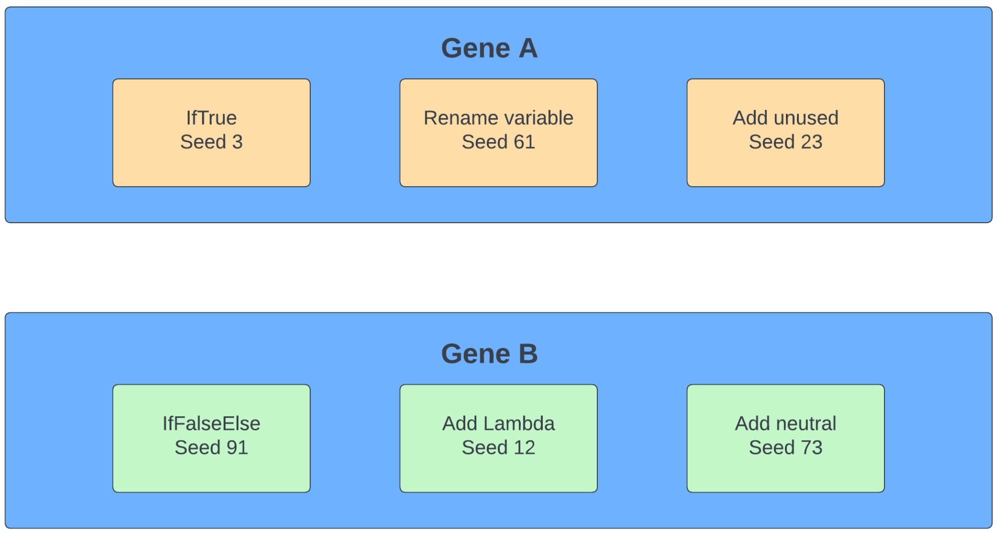


Figure 4.3: Example chromosomes for the genetic algorithm

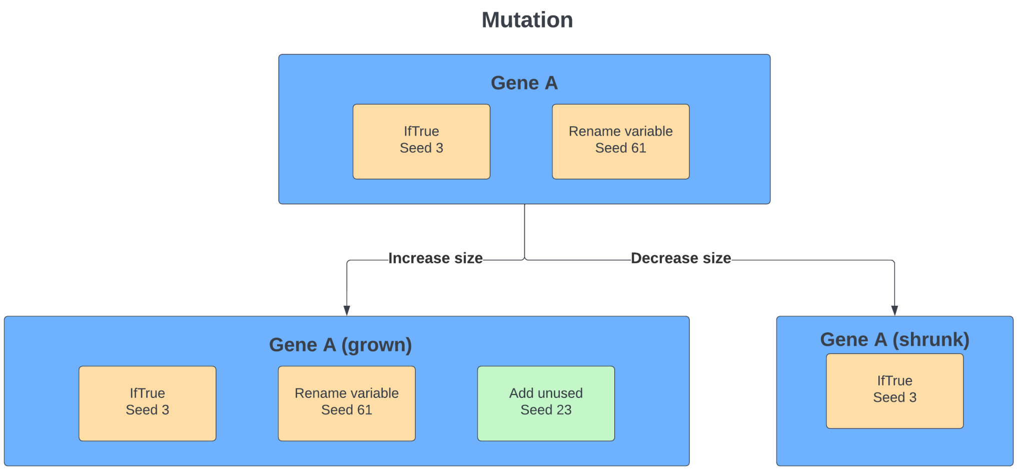


Figure 4.4: Example mutation of genes for the genetic algorithm

of generations. This will usually be between 20 and 50 for population sizes of around 10. If the best fitness improves, this is reset back to zero. This way, we try to ensure we do not get stuck in a local top and potentially mutate our way out if we are stuck there. All these things used for the evolution of the population are done with a certain probability. In projects that use genetic search, there is usually a time-based second termination criterion. If the program exceeds a certain time threshold, it terminates and is considered complete. This thesis will also have such a time threshold.

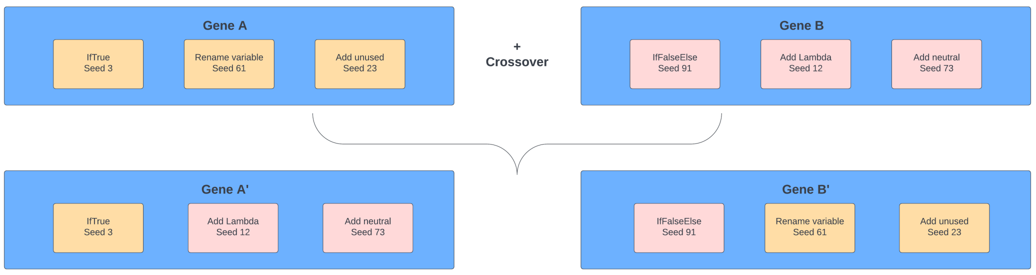


Figure 4.5: Example crossover of genes for the genetic algorithm

4.4 Model architecture

In this thesis, we use the code2vec neural network[3] and evaluate their trained model. However, to get a clear picture of how all the experiments work, we should look at the architecture that Alon et al. used. This architecture is depicted in Figure 4.6. To explain this, we distinguish three parts combined in the figure: the attention weights, the context vectors, and the prediction. The background explains the attention mechanism (Chapter 2). This architecture's main idea is to compute a scalar weight for all combined context vectors. Just like most learnable aspects of a neural network, it is initialized randomly and learns simultaneously with the rest of the network.

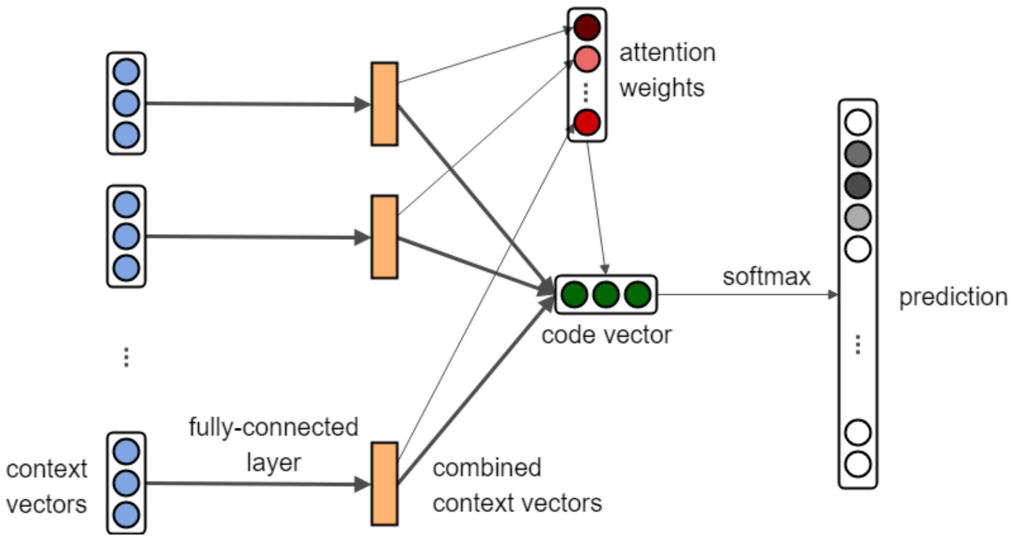


Figure 4.6: Code2Vec neural network architecture (Alon et al. 2019)

Next are the context vectors. All the context values are learned, and the values represent the semantic meaning of the context and the attention the context should get. The problem Alon et al. ran into was that there could be an arbitrarily large number of context vectors, and all of them need to be aggregated into a single vector. A single context vector consists

of three items, two value embedding vocabularies (*value_vocab*), and a path embedding vocabulary (*path_vocab*). The *path_vocab* maps the two *value_vocab* values to each other.
The fully connected layer depicted in the figure learns to combine the different vector components. The model can give different attention to each combination of path and value embeddings. A certain path can get more attention with some values and less attention with others.
The next step in the context vector is aggregating multiple contexts into a single vector representation together with attention. The attention weight a_i for a corresponding context vector is calculated as follows:

$$a_i = \frac{exp(c_i^T \cdot a)}{\sum_{j=1}^{n} exp(c_j^T \cdot a)}$$

This equation means that the sum of the attention weights is always 1, as standard in softmax functions. The code vector v is then simply taken as $\sum_{i=1}^{n} a_i \cdot c_i$. As explained in Section 2.2, a simpler way of looking at attention is as a weighted average, where the weights are learned with respect to all other path contexts. A more detailed explanation of the attention mechanism in the code2vec models is given in Section 2.2. As mentioned in this section, our program uses soft attention. The attention used in the architecture depicted in Figure 4.6 is also explained in more detail.
The last part is the prediction and is essentially the probability that a specific tag is assigned to a given code snippet. The prediction is calculated by computing the softmax-normalized dot product between the code vector and each of the tag embeddings. These tag embeddings are learned as part of the model training.

4.5 Fitness calculation

This research aims to find a variance in the different metrics used to represent the performance of trained code2vec models. With the known metrics, like MRR and F1-score, we can compare the baseline to the final results and see the variance. If this variance is large, the model is not robust.
This research will also aim to optimize the multi-objective performance of metrics used in prior works and custom metrics. We divide our metrics into primary and secondary metrics. The primary metrics are metrics that we will try to optimize through genetic search. The secondary metrics are used to display different performance statistics. The primary metrics this research will look at are:

- MRR
- F1-score
- Recall
- Precision
- Edit distance

The metrics edit distance is calculated over the first prediction of the model. The edit distance metric score is calculated by $1/(edit\_distance+1)$. The edit distance is taken between the prediction of the model with the highest likelihood and the actual label.
Besides those, we also have a custom metric called percentage_MRR. This metric is used to answer the third research question. The percentage_MRR metric takes the certainty of a model's prediction into account instead of only considering the rank, like with MRR. An example of the calculation is when the certainty of the correct prediction is 20%, the score for that prediction is $\frac{20}{100}$. This penalizes the model when it is uncertain and rewards a certain prediction. For all predictions the model makes, we average the score like MRR. The equation for this and MRR are depicted in Figure 4.7. For example, when we take predictions [monkey, gorilla, cat, dog] again with the respective certainties of [35%, 20%, 15%, 10%] where the correct answer is cat. The MRR score is $\frac{1}{3}$ whilst the percentage_MRR score is $\frac{15}{100}$. This is an example with just one data point. The program reports the average percentage_MRR score when there are multiple data points.

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_i}$$

$$Percentage_{MRR} = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{certainty_i}$$

Figure 4.7: Equations for MRR and Percentage_MRR

We define secondary metrics as those that do not directly show the performance of a model. The most important secondary metric we inspect is the number of transformers, as we expect that the model's performance will decrease with more transformers applied. With this secondary metric, we can, for example, look for the biggest change with the least amount of transformations.
Other than that, we also look at the prediction length and the input length as secondary metrics. The prediction length is determined by taking the average length of the predictions. It is used to see if we can simultaneously maximize and minimize the prediction length while maximizing performance metrics. The input length is calculated the same way.
These metrics are interesting because we do not want to make the files too big or make them unrealistic. Another reason for keeping track of these metrics is that we need to keep the changes close to the original data points.

4.5.1 Impact of a smaller dataset

In our experiments, we randomly sample 30 files from the java-small dataset that code2vec offers, which resembles 0.58% of the whole test set. The trained model might not perform as well on the randomly selected files as it would on the whole dataset, which might result in a lower performance. For our experiments, we do, however, want a baseline performance of our sample dataset so that we can measure the change. This baseline performance is measured at the beginning of every run on the unedited dataset.

4.5.2 Pareto front

A great way to look at multi-metric performance is the Pareto front, where each cell in a Pareto solution corresponds to the fitness of a metric. The Pareto front allows us to see what the multi-metric optimums are. The Pareto front is based on dominance, defined as a multi-objective solution where no part of the solution is worse, and at least one is better. So the solution $(1,2)$ is Pareto dominant over $(1,1)$ but not dominant over $(2,1)$. A solution is added to the Pareto front when it is Pareto efficient, meaning that no other solution in the front dominates it. A solution is Pareto efficient if another solution does not exist where an element is better without another element being worse.
For this research, a new solution is added to this set when no metrics lose and at least one improves. This results in a set of solutions defined as the Pareto front, which is a set of efficient choices for our multi-objective problem. This thesis uses the Pareto front for its primary and secondary metrics. Ngatchou et al.[27] give a clear explanation of solving a multi-objective optimization problem. The Pareto front is also explained in more detail in their paper.
To illustrate the theory mentioned here, we look at an example. If the current Pareto front is $[(1,3),(2,2),(3,1)]$ and the goal is to minimize, then adding a solution that is better than one or more of the current ones should change the front. Suppose $(1,1)$ is added as a new solution. In that case, it is better than all the current solutions meaning that the new Pareto front is $[(1,1)]$ because this solution is Pareto dominant over all other solutions. If instead of $(1,1)$ we add $(5,1)$, the Pareto front doesn't change. This solution is not dominant over any of the current solutions on the Pareto front. If instead, the Pareto solutions have more dimensions, the same logic applies. However, in this thesis, we only use two dimensions.

Chapter 5

Experiment Setup

In this chapter, the experimental setup is discussed. These experiments answer the research questions specified in Section 1.2. First, we specify the hardware specifications so that the time each experiment takes can be estimated by people reproducing them. After that, all individual experimental setups and how they answer the research question are mentioned. We ran each experiment 10 times because the project had randomized elements. We take the average of these ten runs and compare that to the baseline performance of the model (performance without any transformations applied). In Figure 5.1, the flow of the experimental setup is displayed. Everything in this diagram will be explained.

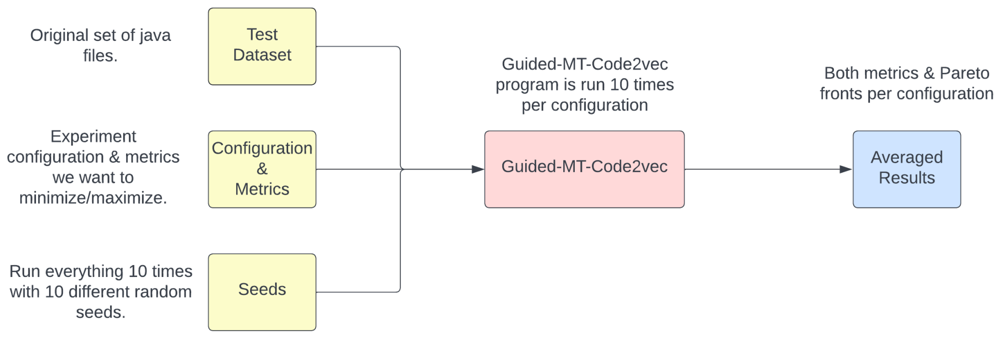

Original set of java files.
Test Dataset
Experiment configuration & metrics we want to minimize/maximize.
Configuration & Metrics
Run everything 10 times with 10 different random seeds.
Seeds
Guided-MT-Code2vec program is run 10 times per configuration
Guided-MT-Code2vec
Both metrics & Pareto fronts per configuration
Averaged Results

Figure 5.1: A diagram for the experiment setup of this thesis

5.1 Computer specifications

The experiments explained in this chapter run on the high-performance SERG (software engineering research group) faculty computer (HPC). The SERG faculty has a share of the machine and a certain number of CPUs that the members of the faculty can use. I ran the experiments on these CPUs to run them in parallel and faster than my machine could.
The computer has an AMD EPYC 7H12 64-Core Processor as a CPU. The experiments run

for this thesis used four cores per experiment, and ten experiments were run in parallel. This means that in total, we used 40 cores, which we could do on one CPU.

5.2 Parameter sampling

This genetic search algorithm in this thesis has multiple parameters that need to be fitted before we can start the experiments. This will be done using the sampling method. With sampling, we try different parameter combinations to see which converges fastest to the optima. There are three parameters for which we chose the values using sampling. These parameters are the uniform rate, the mutation rate, and the increase rate. These rates are explained in Section 4.3.1 and defined below. We picked three potential values for each of the abovementioned rates and sampled all combinations. The outcome of this with the values for all other parameters are:

Hyperparameter	**Value**
Crossover rate	0.7
Mutation rate	0.4
Tournament size	4
Increase rate	0.7
Population size	10
Maximum steady generations	35
Maximum time in minutes	480

Table 5.1: Hyper parameters used in all experiments

Since the fitness function is expensive, we chose a low population size of ten, which still covers a good initial population. Together with this, we chose a tournament size of 4 so that the best individuals would be kept in the population.

5.2.1 Termination parameters

This thesis has two termination parameters: the maximum amount of steady generations and the maximum amount of time in minutes. Both are mentioned in table 5.1 with their respective values. We chose 35 for the maximum amount of steady generations because, during sampling, this value made the program run fast enough. If it does not find a new solution within those 35 generations, it will usually not find it in 40 generations either. Any lower will significantly limit the program's ability to find a better solution.
The maximum time in minutes was also chosen by sampling. The experiments should not exceed 480 minutes, but when they do, we will terminate the program and restart the experiment. During the sampling for the hyperparameters, the most time an experiment took was 240 minutes. This number was doubled to get the maximum time an experiment could take.

5.2.2 From data point to output

In the experiments for this thesis, each data point equals a function in the dataset that is being evaluated. When a metamorphic individual is evaluated, the genotype's metamorphic transformers are applied randomly to each class. This modifies the dataset, which is then used to evaluate the model and output the predictions. From these predictions and corresponding certainties, we can then create a score corresponding to the metric configured for that particular experiment. This score will then be translated to a fitness depending on the number of metrics used in the experiment.

5.3 Experiments

In the experimental setup, we must ensure we can answer all research questions posed in Section 1.2. All experiments described here are divided into the research question they are used for. As mentioned earlier, we did the experiments on the 30 test files we randomly sampled from the java-small dataset. In these 30 files, there are 267 data points(= methods) that the code2vec model evaluates.
To answer the first research question, we need to answer: Does adjusting the semantics of the test-set data affect the scoring metric? To answer this, we will look at the difference between the average fitness without any transformers added and the average of the best fitness of the genetic search runs. This gives us insight into how data augmentation can affect the scoring metrics and to what extent. We try individually minimizing the metrics MRR, F1-Score, recall, and precision. These metrics are all commonly used to represent the performance of the model. These experiments can thus indicate whether there is a significant difference when adding transformers. This is a total of eight experiments to answer this research question. For these eight experiments, there will thus be eight different configuration files that have the respective metrics enabled, and the objective is to minimize the score of this metric through genetic search.

Research question	**Experiments**
Does adjusting the semantics of the test-set data affect the scoring metric?	Perform Genetic Search to Minimize each: MRR, F1, Recall, and Precision

The second research question is: Does adjusting the semantics of the test-set data influence the combination of performance metrics? This question has already been partly answered by the experiments we ran for research question 1. The original performance of the model was measured with the MRR, F1, Recall, and Precision metrics. To see whether the transformations improve the performance of the combinations of these metrics, we will assess the combination of these primary metrics with the objective of minimizing the performance. Recall and precision should be opposing metrics. When the Recall score goes up, the Precision score usually goes down. Optimizing a combination of this will thus be interesting to see if we can find a Pareto front for this. As mentioned above, all individual metrics were already evaluated for research question 1.

We will minimize the number of transformations to see a clear Pareto front between the number of transformations and their respective primary metric. We will see what the Pareto front, explained in Section 4.5.2, is and compare that to the baseline results. The Pareto front will show multiple multi-metric optima between the following combinations of two metrics:

1. Recall & Precision
2. F1 & MRR
3. Recall & number of transformations
4. MRR & number of transformations
5. F1 & number of transformations
6. Precision & number of transformations

The number of transformations is an interesting metric since it shows us how much we need to influence the raw source code before a certain metric is affected. We aim to illustrate this using a graph showing the Pareto front of the metrics combinations mentioned above.

Research question	**Experiments**
Does adjusting the semantics of the test-set data influence the combination of performance metrics?	Search for Trade-offs between combinations of Primary and Secondary Metrics and Display them via a Pareto Front

For the last research question, we try to answer whether there is an alternative metric to MRR that can correctly represent the performance of a model. We formulated an alternative metric to the ones used in other papers. This metric is percentage_MRR as explained in Section 4.5.To see if the percentage_MRR gives a good representation of the performance of the trained model, we evaluate percentage_MRR together with the experiments we used to answer research question 2. We replace MRR with percentage_MRR and look at the discrepancy between the outcomes to see whether percentage_MRR correctly represents the model's performance. We also evaluate percentage_MRR together with MRR and on its own. In every experiment, the metrics are minimized except for the prediction length, which is both minimized and maximized.

1. percentage_MRR & F1
2. percentage_MRR & MRR
3. percentage_MRR

From these experiments, we collect the percentage change between the baseline value and the result and the Pareto fronts of the metric combinations. These Pareto fronts will display a relationship between the metrics for that experiment.

Research question	**Experiments**
Is there an alternative metric to MRR that can correctly represent the performance of a model?	Compare the change values of the new metric to MRR

5.4 Random experiments

For a better insight into the benefit of using guided metamorphic transformations instead, we will also perform random metamorphic transformations on the test set and see the consequences of that. We do this for the one metric experiments and the comparison with the number of transformations. With these extra experiments, we hope to depict the differences between the baseline values, the random experiments, and the genetic search.
This results in a total of 9 extra experiments that are listed below:

1. F1
2. MRR
3. Recall
4. Precision
5. Percentage_MRR
6. Recall & number of transformations
7. MRR & number of transformations
8. F1 & number of transformations
9. Precision & number of transformations

5.5 Significance testing

In software engineering research with machine learning, multiple models or results are often tested against each other. Two papers might have a different neural network for the same task where they want to determine if either is an improvement. Using statistical testing, the authors can determine if a model is significantly better than another model. For this, we first assume the null hypothesis. Under the null hypothesis that both models perform equally well, it can be determined that the results are not likely to occur at random. The p-value is the chance that the null hypothesis would be wrongly rejected. In other words, finding a difference that should not be there. Suppose the results produce a small enough p-value (< 0.01), then it can be concluded that the null hypothesis is incorrect, which means that the models do not perform equally well, and one of them is better. The lower the p-value, the greater the statistical significance of the difference.

5.5.1 Statistical tests

A statistical test that applies is the paired Wilcoxon test[9]. The paired Wilcoxon test is used to determine statistical significance between two dependent sets of values, which is the case here as the same source code is part of both sets. This thesis cannot use the paired t-test because the data is not distributed according to a normal distribution[36].
To perform the paired Wilcoxon test, we first rank the paired differences according to their absolute values. Then, we compute the sum of positive ranks and the sum of negative ranks. The minimum of these will be the test statistic, W_R. When then, fill in the following three formulas:

$$\mu_{W_R} = \frac{n(n+1)}{4}$$

$$\sigma_{W_R} = \sqrt{\frac{n(n+1)(2n+1)}{24} - \frac{\sum t^3 - \sum t}{48}}$$

$$Z_W = \frac{W_R - \mu_{W_R}}{\sigma_{W_R}}$$

In practice, all these formulas are performed using statistical analysis software. This thesis uses the python package *scipy.stats.wilcoxon*[1] for that purpose. The null hypothesis for these tests is that the sets are not statistically significant. When the p-value is less than 0.05, we consider the null hypothesis rejected and thus consider there to be a statistical significance.

5.5.2 Effect size

The effect size is used to determine the magnitude of the results found. This is often used with the p-value to indicate whether the results are statistically significant and if they have a large effect. Sullivan et al. explained very nicely why this is necessary and said: ”Effect size helps readers understand the magnitude of differences found, whereas statistical significance examines whether the findings are likely to be due to chance.”[37].
Together with the paired Wilcoxon test, this thesis uses Cliff’s delta[22] to determine the delta value. According to Vargha et al.[38] the delta value is considered large if it’s greater than 0.43.

[1]https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.wilcoxon.html

Chapter 6

Results

This chapter discusses the results from the experiments discussed in Chapter 5. This chapter is divided into three sections for each type of experiment: first, the individual metric, excluding the custom metric percentage_MRR, experiments will be discussed. Secondly, the multi-objective experiments will be discussed, and the different Pareto fronts will be illustrated. Then, the custom metric experiment results are illustrated. For each section, we will talk about the implications of the results. Lastly, we will look at the distribution of transformers used in the best individuals and see if we can see some pattern for which transformer has the most impact on the metrics.

6.1 Individual metrics

As explained in Section 5.3, the first experiment of this thesis was to evaluate the individual metrics that have been used for several years. The objective was to minimize the following metrics: F_1-score, MRR, recall, and precision. Through this, we try to discover the model's weaknesses and assess our program's impact. The most important thing to look at in the results of these experiments is the initial score and the best score after the genetic algorithm. We ran every experiment 10 times to account for the random factor in the program. We average over the ten runs per experiment and get one average initial and one average maximum value for each experiment. These values are depicted in Figure 6.1 together with the Percentage_MRR metric experiment.
We can see that the average best, which is the best score according to the objective function, is lower than the average initial/baseline values. This means that adding the different transformers decreases the performance of the trained code2vec model. As a reference, we also added random experiments to the histogram. We can see that the random metamorphic transformations already negatively affect the model's performance. This is the case for all individual metrics as displayed in Figure 6.1

In Table 6.1 below, we use these values to determine whether the results we got are significant. We use the paired Wilcoxon test to determine this, and the corresponding p-values are also given in this table. Since all these values are below 0.01, we can conclude that there is a significant drop in performance when metamorphic transformations are added to the dataset that the trained code2vec model evaluates. The same table also indicates the effect size of each of the experiments. The effect size measures the magnitude of the effect of the experiment. The greater this value is the more significant the experiment's impact. All our effect sizes are considered large according to the theory in Section 5.5.2.

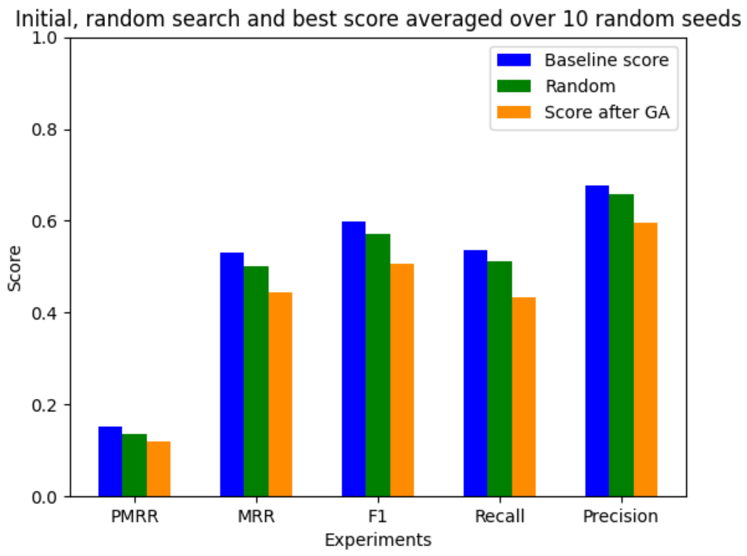


Figure 6.1: PMRR, MRR, F_1-score, recall, and precision metrics initial, random, and after GA scores

Experiment	Baseline	Random	After GA	Wilcoxon p-value	delta value
PMRR	0,152	0,135	0,119	0.00586	0.66
MRR	0,531	0,472	0,445	0,00195	1.0
F1	0,598	0,572	0,506	0,00195	1.0
Recall	0,536	0,511	0,434	0,00195	1.0
Precision	0,678	0,658	0,595	0,00195	1.0

Table 6.1: Individual experiments and corresponding p-values

Adding the random transformations experiment allows us to see the rate at which the performance declines with different amounts of transformations applied. In Figure 6.2 the differences between the initial baseline score, the score with 5, 10, and 20 random transformations, and the score after the genetic algorithm are visualized. This is done for all single metric experiments, and the rest of the graphs can be found in Appendix A.2. For every metric tested, adding more random transformations decreases the score, but to a larger extent for the genetic algorithm.

Since we use different scoring metrics that consider multiple predictions made for one label, it is hard to put this into perspective. However, if we assume that all correct predictions made were the first guess by the model and the rest is wrong, we can say that this MRR score means that about 53% of the predictions are correct. When we add the metamorphic transformations, this drops to 45% correct predictions.

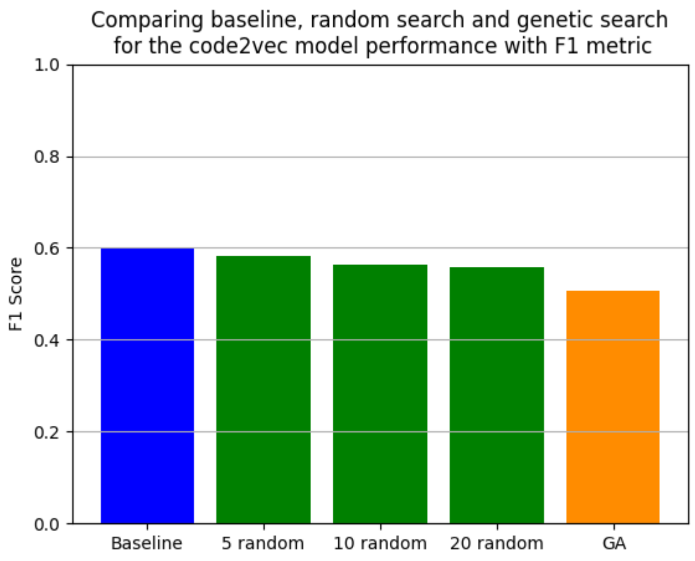


Figure 6.2: Comparison baseline, random search, and genetic search with the F_1-score metric

For all primary metrics, our results show that both random application and genetic search can impact the metric. Our results show that genetic search results in a 12.3% to 21.7% change between the baseline score and the final score

6.2 Multi metric experiments

We also ran experiments taking into account different combinations of metrics. Illustrating these metric combinations will give us insight into the effect of transformations on the metric combinations and whether one metric might be affected more than others. First, we want to examine the impact of the number of transformations on each basic metric. An interesting thing is the extent to which an x amount of transformations that are not random can impact a specific metric. We anticipate that applying more transformations will further decrease the model's performance and that the guided search will result in a bigger performance impact than the random search.
Subsequently, we look at the impact on the combined basic metrics. Here we expect the Pareto fronts to be around the linear line for each combination of metrics. The goal is to see

the trade-off between these metrics and see what metric decreases more in score than the other. The Pareto fronts give a perfect visualization of this.

6.2.1 Number of transformations comparison

In Figure 6.3, we can see how the score changes with every extra transformation we add. However, given the same number of transformations added, the impact on the performance is bigger for the guided search (purple dots) compared with the random search (green dots), as shown by the different shapes and slopes of the fitted lines. To illustrate, the average initial and average scores after the genetic algorithm have also been added to the graphs. For the average score after the genetic algorithm, the average F_1-score and the average number of transformations are used. For the Pareto solutions, we use the median with each number of transformations for both the genetic and the random search.
For the guided search, the number of transformations added initially has a bigger impact on the F_1-score but becomes smaller when multiple transformations have already been added. For the random search, the relationship is more linear, meaning that additional transformations added have a similar impact on the performance.

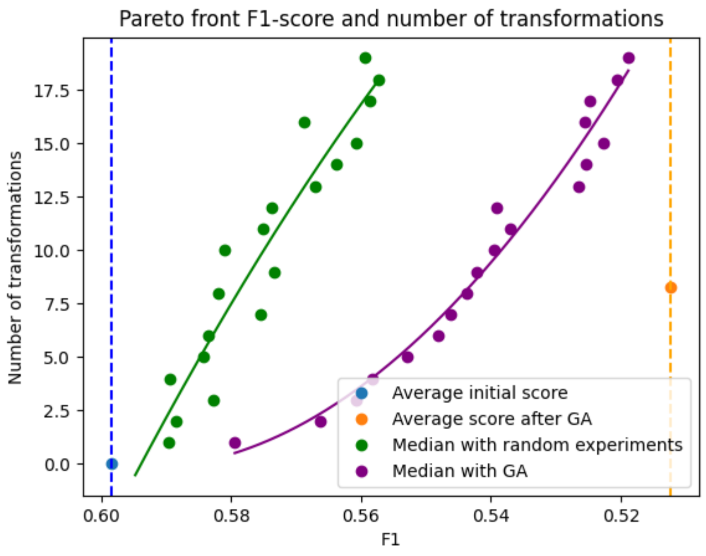


Figure 6.3: F_1-score & number of transformations median Pareto

We come to the same conclusions when we look at the results for the experiments of the other three metrics, MRR, recall, and precision. These graphs can be found in Appendix A.3. We can see that the random experiments result in a performance drop for all multi-metric experiments with the number of transformations. However, the guided experiments have a bigger drop for every number of transformations applied.
From the results in Section 6.1, we already know that certain metrics experienced a bigger

drop than others. For example, MRR was affected more by the genetic algorithm than the F_1-score. In Figure A.14 we see that the drop in performance with each added transformer does not become significantly smaller. Adding more transformations does not decrease its impact on the MRR metric. To conclude this, we need more data points than we currently have.
We see a different trend in the recall metric, Figure A.12. The trend for the recall metric is more linear. In both the genetic search and the random search, we see an increase in the impact of each transformation when more transformations are added. We also notice more outliers than in the previous two graphs.
Figure A.13 compares the precision score with the number of transformations. We notice that the average score after the genetic algorithm is no longer the lowest value in the graph. This graph clearly illustrates the difference between taking the average over multiple runs and taking the median. There is a median point that resulted in a worse precision score. However, some runs finished with a higher score meaning that the average score after the genetic search runs is also a little higher. We also see that the impact of added transformations by the genetic algorithm becomes larger. From this, we can conclude that when a more significant number of transformations are applied, they have a greater effect on the precision metric than, for example, the recall metric.

6.2.2 Basic metric combinations

The first multi-basic-metric experiment we did with the genetic algorithm was recall and precision. As mentioned in Section 2.8, some predictions can have a low recall but a high precision score. An example of this is when the correct label is *CountLines* and the prediction is *Count*. This will get a precision score of 1 since *Count* is in the expected labels, but a recall score of 0.5 since *Lines* is not in the predicted labels. Because of this, we assumed that both would decrease at a similar rate when transformations were added. We can see in Figure 6.4 that there is indeed a linear relationship between these metrics when metamorphic transformations are applied.

If we reverse engineer the F_1-score to its solution set of recall and precision values, we get a quarter circle in our graph, which is visualized in Figure 6.5. Here we can see that the solutions gathered by the multi-objective Pareto front are in line with the solution circle for the average F_1-score after the genetic algorithm. From this graph, we can conclude that the single-metric experiments for the F_1-score behave similarly to the projected behavior in the Pareto front of this experiment.

Another interesting combination is the two most used metrics for measuring performance in method name prediction. With previous results, we have already seen the impact of metamorphic transformations on both of the individual metrics. Through the individual experiments, we already established that the performance decreases by adding metamorphic transformations to the test dataset.
In Figure 6.6 the points are more on a constant line where MRR slightly decreases but F1 stays rather constant. When the $(0,0)$ point has been excluded, the trendline (dotted line)

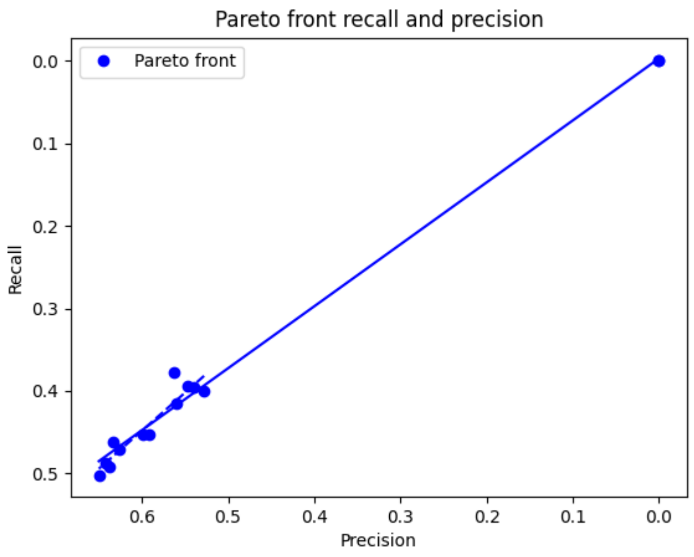


Figure 6.4: Recall and precision metrics Pareto front measured with genetic search

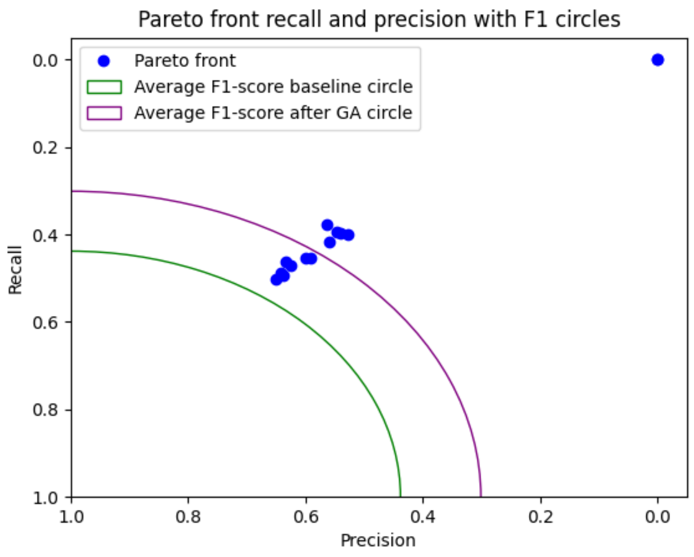


Figure 6.5: Recall and precision metrics Pareto front measured with genetic search with F_1-score circles

changes to a more horizontal line. This means that both metrics are impacted when metamorphic transformations are applied. However, we can see that the MRR is impacted more than the F_1-score when we optimize both metrics. This is consistent with the single metric experiments and thus was the expected result.

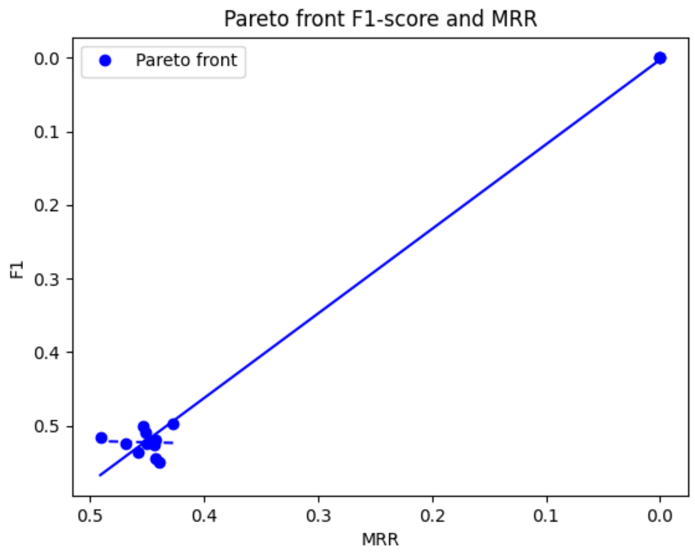


Figure 6.6: F_1-score and MRR metrics Pareto front measured with genetic search

Our results show that the performance drop decreases for all metric combinations when more transformers are added. The results show a linear relationship in the Pareto front of the metric combinations. However, some metrics suffer more from the applied transformations than others

6.3 Percentage_MRR

In section 4.5 the custom metric percentage_MRR was proposed as a new metric that would better reflect the performance of a model. Percentage_MRR penalizes the certainty of a model instead of penalizing the prediction rank. We already saw the performance decline of percentage_MRR together with the other metrics in Figure 6.1.
In Table 6.2 this metric is compared to MRR. We can see that the initial score is less than a third of the MRR score, which was expected since percentage_MRR is a stricter metric. One notable thing is that the relative difference for percentage_MRR is higher than that of MRR. This means that the impact of metamorphic transformations is higher for this metric than for the MRR metric, which indicates that the model becomes more unsure with metamorphic transformations added. All of this is visualized in Figure 6.7.

Experiment	Baseline	After GA	Relative difference
MRR	$0,531$	$0,445$	$-16,14\%$
Percentage_MRR	$0,152$	$0,119$	$-21,74\%$

Table 6.2: Individual experiments and corresponding relative difference

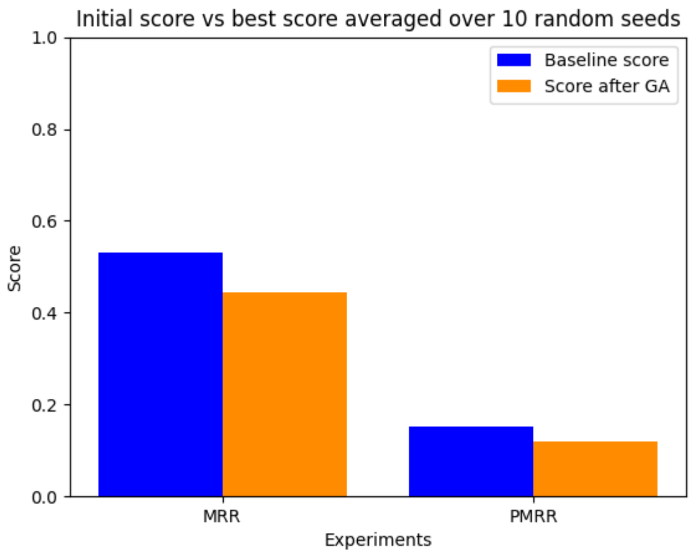


Figure 6.7: Comparing MRR and percentage_MRR their baseline score and score after the genetic algorithm

For further comparison, we displayed the density graphs of both metrics, Figure 6.8. We see that the distributions of both metrics are similar, except for the fact that the percentage_MRR distribution is diluted compared to MRR. It is, however, important to keep in mind that the main difference between these metrics is that the percentage_MRR is a stricter metric, meaning that the score will always be lower. If the resulting percentage_MRR score is higher, meaning that the model is certain about its predictions, and adding MTs will likely not change too much. This means that to get a more accurate representation of the robustness of the model, percentage_MRR could be used.

The next metrics we did the multi-metric experiment with were percentage_MRR and MRR, Figure 6.9. This experiment aimed to see whether both metrics would decrease at around the same rate when transformations were applied to the test dataset. Like with the other metric combinations, there is a linear relationship between percentage_MRR and MRR when the genetic algorithm uses both metrics in the objective function with equal weights. We also evaluated percentage_MRR together with the F_1-score. However, there was no clear relationship in the Pareto front of these two metrics, as seen in Appendix A.3.

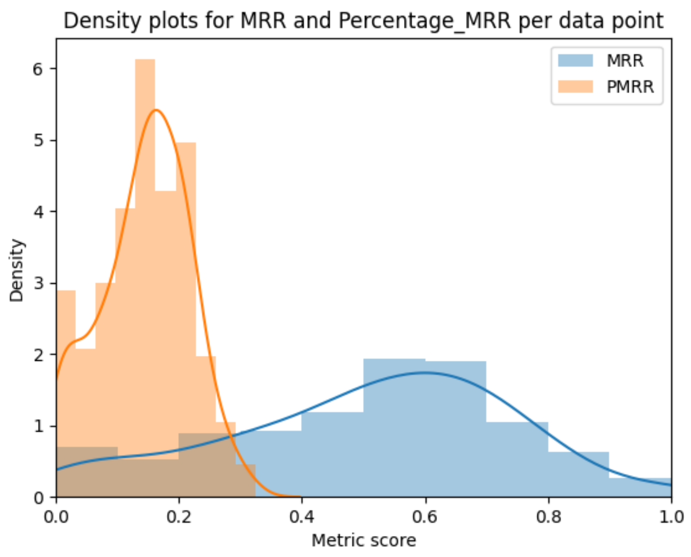


Figure 6.8: Comparing the density graphs of MRR and percentage_MRR

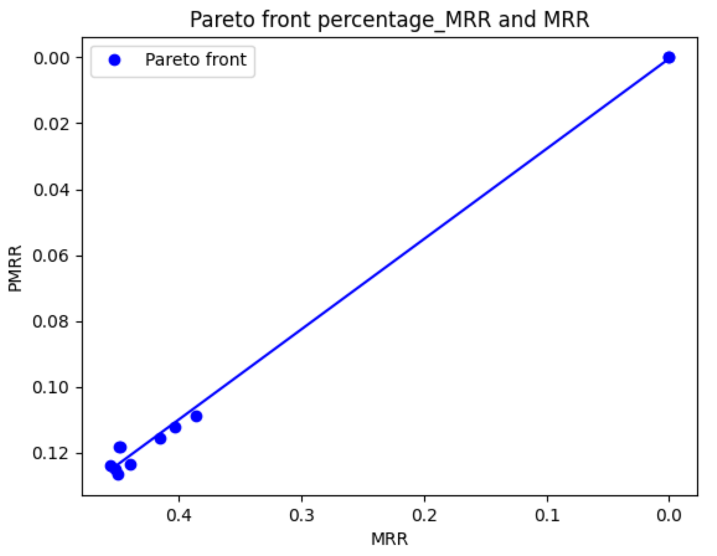


Figure 6.9: Percentage_MRR and MRR metrics Pareto front measured with genetic search

The proposed metric, percentage_MRR, can be considered an alternative to MRR, which is catered toward determining the certainty of the model. It also suffers a bigger drop in performance score when metamorphic transformations are applied using genetic or random search. These observations indicate that percentage_MRR is a good robustness metric that penalizes uncertainty.

6.4 Transformation distribution

As stated in Section 4.3.1, there was no limitation on the occurrence of different transformers. This means that the expectation was that the transformer with the most impact on the performance would occur most. These transformer distributions can give us insight into which transformations have the biggest impact on the performance and whether there are any trends in these distributions.
We investigated which MTs had the most impact on the performance of the trained code2vec model by measuring the final best individuals of each run in the experiment. This is depicted in a histogram in Figure 6.10. We can see that there are certain transformers, such as the *IfFalseElseTransformer*, has more impact than, for example, the *IfTrueTransformer*. This means that the trained model is less robust against certain types of transformers than others. These findings are relevant to making these models more robust.
We predicted that the transformer that added the most code would be the one that would end up in the most final individuals and thus be the most impactful. The if-false-else transformer aligns with this observation since it adds quite a lot of code to the methods. However, according to this logic, the if-true transformer should also be influential to the performance of the trained code2vec model. However, this transformer was not in a lot of final individuals. This could be an exception to the rule since the lambda transformer was very influential and makes quite some changes to the underlying AST. As expected, the rename variable transformer did not end up in many final individuals since this only changes the underlying AST in a minimal way.

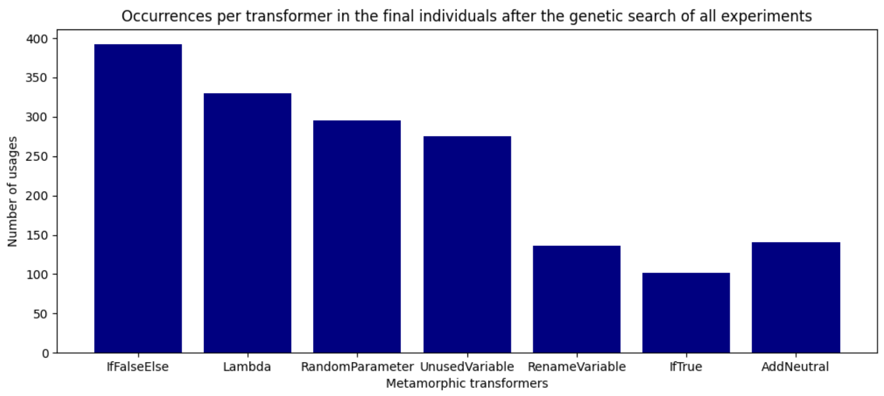


Figure 6.10: The distribution of transformers in the final individuals after the genetic algorithm

We can then compare the transformer distribution from Figure 6.10 to the results the Lampion paper[4] published. The Lampion paper reported that the most impactful transformers added an unused variable name, while the if-true and if-false-else transformers were the least impactful. This was, however, for a different task than method name prediction using a different scoring metric. Besides that, Applis et al. also combined the if-true and if-false-else transformer in their results, which means we do not know whether the if-false-else transformer was just as effective in their experiments as it was in ours.

6.4.1 percentage_MRR vs MRR

In this section, we look at the transformer distribution for all experiments of the MRR metric compared to that of the percentage_MRR metric. This can indicate which transformers have the biggest impact on these metrics and how they differ.
In Figure 6.11 both the transformer distribution of the MRR and percentage_MRR metrics are depicted for comparison. For the MRR metric, we see that the if-false-else transformer is the one that appears most in the final individuals, which is in line with the total transformer distribution. The random parameter transformer, however, appears the least, contrary to what Figure 6.10 shows.
When we compare this to the percentage_MRR transformer distribution, we can see quite a few differences. For instance, the if-false-else transformer is one of the least impactful for this metric, and so is the unused variable transformer. The other contrast is that the if-true transformer is very impactful and occurs in quite some final individuals.

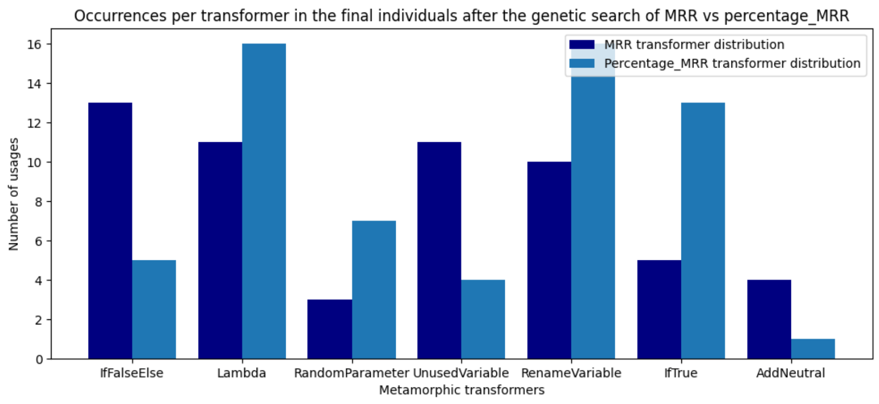


Figure 6.11: The distribution of transformers in the final individuals after the genetic algorithm for the MRR experiments versus the percentage_MRR experiments

From these figures, we see that there are quite some differences between the distributions. These results, however, are based on ten experiments, which means that we cannot say for certain that these results will be the same when we increase the number of experiments to 100 or even more. This might thus be something a future work could look at.

Chapter 7

Discussion

In this chapter, we discuss the validity of this research and how the tool created could be applied in the development phase of different projects. We also look at the consequences of the results found in this thesis and what it means for people using code2vec or other models. As stated in Chapter 1, the use case of the tool created with this thesis is that researchers would use it while evaluating their models. They could run their model against this tool and evaluate how robust their model is against metamorphic transformations. They could also use this tool to evaluate whether specific changes in the training have made a positive impact on the vulnerability against metamorphic transformations.
The next question is, when do we stop searching for cases where the model does not perform well? A search algorithm, such as the one used in this tool, is designed to find complex examples for the model. Even if a new model is developed more robust against the metamorphic transformations we used in this tool, it does not mean that there are no other exceptions where the model might perform worse than before.

7.1 Threats to validity

In this section, specific threats to the validity of this work are discussed. People could make these claims as a counter to this thesis. Underneath each subsection is the threats' responses and everything we did as precautions.

7.1.1 The program could have bugs

This means that there could be problems in the results that would threaten the validity. To prevent this, we first made sure that the entire project and experiments were open source. Meaning, that anyone who wants to check the code or rerun the experiments can do this through the docker build. We also made sure that everything was tested and ran the program multiple times in controlled environments to test everything.

7.1.2 Cherry picked files

To prevent this, we randomly picked all thirty files with a script available in the open-source repository. This means that this is reproducible and that anyone that runs this script will also get thirty random files from the entire dataset.

7.1.3 Reproducing it would not yield statistical significance

To minimize this risk, we ran the program with ten random seeds and averaged our results over them. This should minimize the risk of someone else who runs the program getting very different results. Besides that, we tried to adhere to the best practices while writing this thesis and performing the experiments, such as the statistical analysis and displaying.

7.1.4 Transferability of the results

Currently, this thesis uses a small portion of the java medium test set code2vec provided on their Github[1]. Using the whole dataset might result in a different performance drop than this thesis measured. However, due to the computing power and time constraint, we could not perform the experiments with the whole dataset in this thesis.

7.2 Applications

With this tool, performance is expected to drop when a model is evaluated. However, we want the research field to be aware of the drop in performance in these cases and strive for this drop to no longer be significant with newly developed models. This would improve the overall performance and robustness of the models and might even allow these models to be used in the development phase of open source projects. That should be the current goal of the model developers and the way to use this tool. The time to stop searching for exceptions where the model does not perform well is when the model can be properly used as a static analysis tool in the development phase of new projects.

7.2.1 CI/CD pipeline

The program in this thesis can be integrated into a CI/CD pipeline. This would mean that new models, or new developments of an existing model, get evaluated by this tool to ensure that the model's quality does not decrease by too much. The developers could set the threshold manually, giving them the ability to be stricter or looser with their requirements. The developers could also pick the metrics for which they want to run the checks and how many random runs they want to do.

[1]https://github.com/tech-srl/code2vec#additional-datasets

Chapter 8

Conclusions and Future Work

8.1 Research questions

This thesis aims to answer the three following research questions:

1. To what extent does adjusting the semantics of the test-set data using metamorphic transformations with genetic algorithms affect the scoring metric?

2. To what extent does adjusting the semantics of the test-set data using metamorphic transformations with genetic algorithms influence the combination of additional metrics?

3. Is there an alternative performance robustness metric to MRR that can correctly represent the performance of a model?

The first experiments we did were aimed at the individual metrics that have already been used by prior works to establish the performance of their models. These metrics were the F_1-score, MRR, recall, and precision. Through the results of this thesis, we know that the trained models are not robust against other metamorphic transformations. The results indicate that adjusting the test-set data using metamorphic transformations through a genetic algorithm greatly affects the scoring metrics. An example of this is the case of MRR, where the average performance decrease is $16,14\%$. All single metric experiments resulted in a statistically significant performance change and large effect size. Thus, we can conclude that adding metamorphic transformations to the test set significantly affects the scoring metrics.

To answer the second research question, we looked at different combinations of metrics and evaluated them based on the relationship between the points in the Pareto front. The first combinations we looked at all included the number of transformations as a second metric. These results indicated that when the number of applied metamorphic transformations increases, the effect on the performance decreases. This is the case for all basic metrics mentioned earlier. We also conclude that adding random transformations does not impact the performance as much as applying the transformations using the genetic algorithm for

these experiments.
Next, we looked at the combination recall & precision and the combination F_1-score and MRR. For these experiments, we concluded that the impact on the metrics was similar, as expected. Applying the metamorphic transformations to the test set using a genetic algorithm decreases the performance of both metrics by a similar magnitude.

Lastly, we evaluated the custom metric percentage_MRR. The well-known metric MRR inspires this metric. To evaluate whether this would be a good metric to use or whether or not it might be a better metric to assess the quality of a model, we compare it to the experiments where MRR was used. First, we see that the baseline value of percentage_MRR is already much lower than that of MRR. This was expected since we penalize the certainty of the model instead of just looking at the rank. Because of this, we should not look at the baseline value but rather look at the difference the program had on the performance. We saw that the difference between the baseline score and the score after the genetic algorithm was less with percentage_MRR than with MRR, which means that the drop in certainty is less than the drop in rank when more metamorphic transformations are applied.
We then look at the Pareto front of the multi-metric experiment with percentage_MRR and MRR. Here we saw that the relationship between these metrics was linear, meaning both metrics decreased at a similar rate when transformations were applied. We can, however, see that the points deviate a little from the linear line in favor of percentage_MRR. This indicates that percentage_MRR is slightly more robust against metamorphic transformations than MRR. With these findings, this thesis can conclude that percentage_MRR is a good alternative performance robustness metric to MRR and can correctly represent the performance of a trained model. It is, however, a stricter metric.

Given this conclusion and the results that this thesis reported, we can now put it into the context of already developed models. These models can use the tool in this thesis to evaluate their model's robustness against metamorphic transformations. If the model is not robust against these transformations, it does not mean that the model can no longer be used. It does, however, mean that when the model evaluates code that is not written as well as the training examples used, it might affect the predictions the model makes. When using the current code2vec model, the same logic applies.

8.2 Future work

These future work avenues indicate how another work can build on this thesis. Finally, we make an additional remark about the newly introduced metric.

Future of percentage_MRR

The newly introduced metric percentage_MRR is found to be a good performance metric that takes into consideration the certainty of the models about a particular prediction. When these models get used in the software engineering cycle, the models need to be able to predict the method names with a higher certainty before they are considered good. However, since it is a new metric, there need to be works that determine what a good percentage_MRR score is. This has to be done by evaluating different known models with percentage_MRR.

Education

Metamorphic transformations and metamorphic testing are both subjects that are part of different study programs. This tool could illustrate the importance of robustness in machine learning models and how metamorphic testing works. The students could play around with it and determine the impact of different metamorphic transformers on different metrics. In most cases learning while doing is a better way for students to grasp a subject.

Relationship between transformers and the performance

This thesis did not investigate the relationship between different transformers and the performance drops. This could be an interesting future work where a researcher determines why certain transformations have a bigger impact than others. This might result in a better understanding of where the models are currently flawed and what would be possible avenues to improve the models further.
A logical trend here would be that adding transformers that add more code to the methods would result in a bigger performance decrease. However, if another trend can be found, we might see that the current predictions heavily rely on a certain part of the method.

Bibliography

[1] Uri Alon et al. "A General Path-Based Representation for Predicting Program Properties". In: *SIGPLAN Not.* 53.4 (2018), pp. 404–419. ISSN: 0362-1340. DOI: 10.1145/3296979.3192412. URL: https://doi.org/10.1145/3296979.3192412.

[2] Uri Alon et al. "code2seq: Generating sequences from structured representations of code". In: *arXiv preprint arXiv:1808.01400* (2018).

[3] Uri Alon et al. "code2vec: Learning distributed representations of code". In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), pp. 1–29.

[4] L.H. Applis, A. Panichella, and A. van Deursen. "Assessing Robustness of ML-Based Program Analysis Tools using Metamorphic Program Transformations". In: *IEEE/ACM International Conference on Automated Software Engineering - NIER Track*. IEEE / ACM, Sept. 2021.

[5] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural machine translation by jointly learning to align and translate". In: *arXiv preprint arXiv:1409.0473* (2014).

[6] Simon Butler et al. "Exploring the influence of identifier names on code quality: An empirical study". In: *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE. 2010, pp. 156–165.

[7] Simon Butler et al. "Relating identifier naming flaws and code quality: An empirical study". In: *2009 16th Working Conference on Reverse Engineering*. IEEE. 2009, pp. 31–35.

[8] Rhys Compton et al. "Embedding java classes with code2vec: Improvements from variable obfuscation". In: *Proceedings of the 17th International Conference on Mining Software Repositories*. 2020, pp. 243–253.

[9] William Jay Conover. *Practical nonparametric statistics*. Vol. 350. john wiley & sons, 1999.

[10] Manoranjan Dash and Huan Liu. "Feature selection for classification". In: *Intelligent data analysis* 1.1-4 (1997), pp. 131–156.

[11] Michael Denkowski and Alon Lavie. “Meteor universal: Language specific translation evaluation for any target language”. In: *Proceedings of the ninth workshop on statistical machine translation*. 2014, pp. 376–380.

[12] Anurag Dwarakanath et al. “Identifying implementation bugs in machine learning based image classifiers using metamorphic testing”. In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2018, pp. 118–128.

[13] Zhangyin Feng et al. “Codebert: A pre-trained model for programming and natural languages”. In: *arXiv preprint arXiv:2002.08155* (2020).

[14] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. “Structured neural summarization”. In: International Conference on Learning Representations, 2019.

[15] Stephanie Forrest et al. “A genetic programming approach to automated software repair”. In: *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*. 2009, pp. 947–954.

[16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[17] S Delphine Immaculate, M Farida Begam, and M Floramary. “Software bug prediction using supervised machine learning algorithms”. In: *2019 International conference on data science and communication (IconDSC)*. IEEE. 2019, pp. 1–7.

[18] Gareth James et al. *An Introduction to Statistical Learning: with Applications in R*. Springer, 2013. URL: `https://faculty.marshall.usc.edu/gareth-james/ISL/`.

[19] Hong Jin Kang, Tegawendé F Bissyandé, and David Lo. “Assessing the generalizability of code2vec token embeddings”. In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2019, pp. 1–12.

[20] Memoona Khanum et al. “A survey on unsupervised machine learning algorithms for automation, classification and maintenance”. In: *International Journal of Computer Applications* 119.13 (2015).

[21] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of Massive Datasets*. 2nd. USA: Cambridge University Press, 2014. ISBN: 1107077230.

[22] Guillermo Macbeth, Eugenia Razumiejczyk, and Rubén Daniel Ledesma. “Cliff’s Delta Calculator: A non-parametric effect size program for two groups of observations”. In: *Universitas Psychologica* 10.2 (2011), pp. 545–555.

[23] Rohan Reddy Mekala et al. “Metamorphic detection of adversarial examples in deep learning models with affine transformations”. In: *2019 IEEE/ACM 4th International Workshop on Metamorphic Testing (MET)*. IEEE. 2019, pp. 55–62.

[24] Tomas Mikolov et al. “Distributed representations of words and phrases and their compositionality”. In: *Advances in neural information processing systems* 26 (2013).

[25] Tomas Mikolov et al. “Efficient estimation of word representations in vector space”. In: *arXiv preprint arXiv:1301.3781* (2013).

[26] Christian Murphy, Gail E Kaiser, and Lifeng Hu. “Properties of machine learning applications for use in metamorphic testing”. In: (2008).

[27] Patrick Ngatchou, Anahita Zarei, and A El-Sharkawi. “Pareto multi objective optimization”. In: *Proceedings of the 13th International Conference on, Intelligent Systems Application to Power Systems*. IEEE. 2005, pp. 84–91.

[28] Kishore Papineni et al. “Bleu: a method for automatic evaluation of machine translation”. In: *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 2002, pp. 311–318.

[29] Md Rafiqul Islam Rabin and Mohammad Amin Alipour. “Evaluation of generalizability of neural program analyzers under semantic-preserving transformations”. In: *arXiv preprint arXiv:2004.07313* (2020).

[30] Md Rafiqul Islam Rabin et al. “On the generalizability of Neural Program Models with respect to semantic-preserving program transformations”. In: *Information and Software Technology* 135 (2021), p. 106552.

[31] Colin Raffel and Daniel PW Ellis. “Feed-forward networks with attention can solve some long-term memory problems”. In: *arXiv preprint arXiv:1512.08756* (2015).

[32] Veselin Raychev, Martin Vechev, and Andreas Krause. “Predicting Program Properties from ”Big Code””. In: *SIGPLAN Not.* 50.1 (2015), pp. 111–124. ISSN: 0362-1340. DOI: `10.1145/2775051.2677009`. URL: `https://doi-org.tudelft.idm.oclc.org/10.1145/2775051.2677009`.

[33] Payam Refaeilzadeh, Lei Tang, and Huan Liu. “Cross-validation.” In: *Encyclopedia of database systems* 5 (2009), pp. 532–538.

[34] Stefan Sette and Luc Boullart. “Genetic programming: principles and applications”. In: *Engineering applications of artificial intelligence* 14.6 (2001), pp. 727–736.

[35] Ritika Singh and Satwinder Singh. “Text Similarity Measures in News Articles by Vector Space Model Using NLP”. In: *Journal of The Institution of Engineers (India): Series B* 102.2 (2021), pp. 329–338.

[36] Mark D Smucker, James Allan, and Ben Carterette. “A comparison of statistical significance tests for information retrieval evaluation”. In: *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*. 2007, pp. 623–632.

[37] Gail M Sullivan and Richard Feinn. “Using effect size—or why the P value is not enough”. In: *Journal of graduate medical education* 4.3 (2012), pp. 279–282.

[38] András Vargha and Harold D Delaney. “A critique and improvement of the CL common language effect size statistics of McGraw and Wong”. In: *Journal of Educational and Behavioral Statistics* 25.2 (2000), pp. 101–132.

[39] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems* 30 (2017).

[40] Xiaoyuan Xie et al. “Testing and validating machine learning classifiers by metamorphic testing”. In: *Journal of Systems and Software* 84.4 (2011), pp. 544–558.

[41] Kelvin Xu et al. “Show, attend and tell: Neural image caption generation with visual attention”. In: *International conference on machine learning*. PMLR. 2015, pp. 2048–2057.

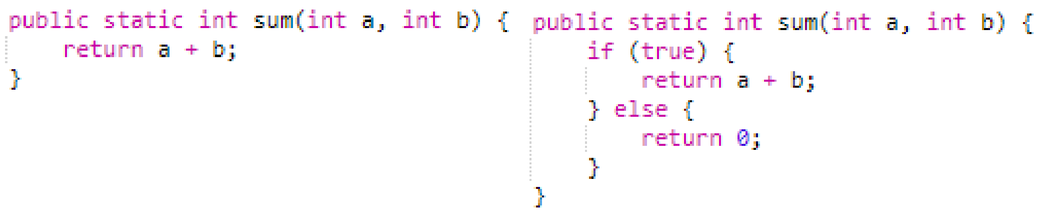
[42] Aston Zhang et al. “Dive into Deep Learning”. In: *arXiv preprint arXiv:2106.11342* (2021).

[43] Zhi Quan Zhou et al. “Metamorphic testing and its applications”. In: *Proceedings of the 8th International Symposium on Future Software Technology (ISFST 2004)*. Software Engineers Association Xian, China. 2004, pp. 346–351.

Appendix A

Appendix

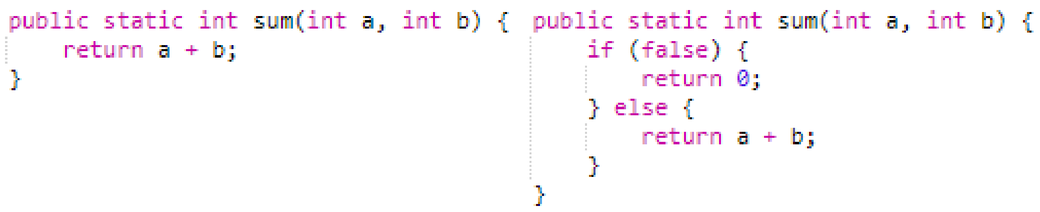
A.1 Transformer examples



```java
public static int sum(int a, int b) {
    return a + b;
}
```

```java
public static int sum(int a, int b) {
    if (true) {
        return a + b;
    } else {
        return 0;
    }
}
```

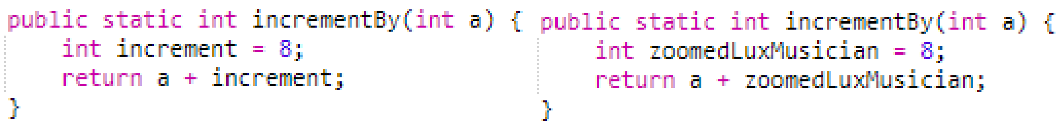
Figure A.1: If-True transformer applied to a sum function. The normal function is on the left, and the transformed function is on the right



```java
public static int sum(int a, int b) {
    return a + b;
}
```

```java
public static int sum(int a, int b) {
    if (false) {
        return 0;
    } else {
        return a + b;
    }
}
```

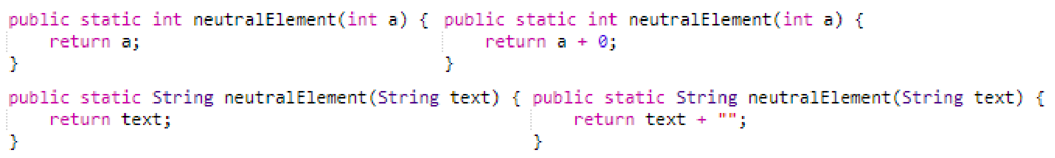
Figure A.2: If-False-else transformer applied to a sum function. The normal function is on the left, and the transformed function is on the right



```java
public static int incrementBy(int a) {
    int increment = 8;
    return a + increment;
}
```

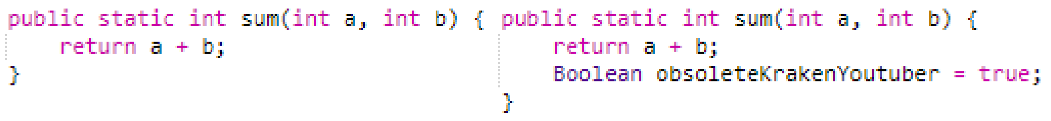
```java
public static int incrementBy(int a) {
    int zoomedLuxMusician = 8;
    return a + zoomedLuxMusician;
}
```

Figure A.3: The rename variable transformer applied to a function. The normal function is on the left, and the transformed function is on the right. The local variable *increment* is renamed to *zoomedLucMusician*



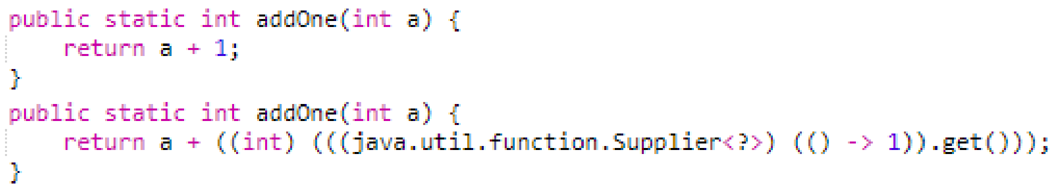
```
public static int neutralElement(int a) { public static int neutralElement(int a) {
    return a;                                 return a + 0;
}                                         }
public static String neutralElement(String text) { public static String neutralElement(String text) {
    return text;                                       return text + "";
}                                                  }
```

Figure A.4: The add a neutral element transformer applied to a function. The normal function is on the left, and the transformed function is on the right. The top example is of the Integer datatype, and the lower one is of the String datatype



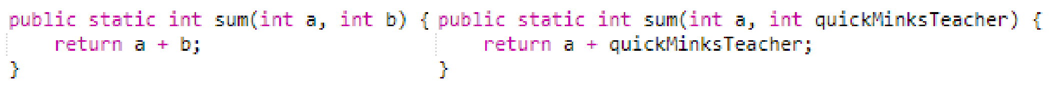
```
public static int sum(int a, int b) { public static int sum(int a, int b) {
    return a + b;                         return a + b;
}                                         Boolean obsoleteKrakenYoutuber = true;
                                      }
```

Figure A.5: The add an unused variable transformer applied to the sum function. The normal function is on the left, and the transformed function is on the right. Here a random variable is added that is never used



```
public static int addOne(int a) {
    return a + 1;
}
public static int addOne(int a) {
    return a + ((int) (((java.util.function.Supplier<?>) (() -> 1)).get()));
}
```

Figure A.6: The lambda identity transformer applied to the addOne function. The normal function is on the left, and the transformed function is on the right. the *1* is changed to a lambda function which still means *1*



```
public static int sum(int a, int b) { public static int sum(int a, int quickMinksTeacher) {
    return a + b;                         return a + quickMinksTeacher;
}                                     }
```

Figure A.7: The Rename parameter transformer applied to the sum function. The normal function is on the left, and the transformed function is on the right. The variable *b* is changed to *quickMinksTeacher*

A.2 Single metric random vs guided experiments

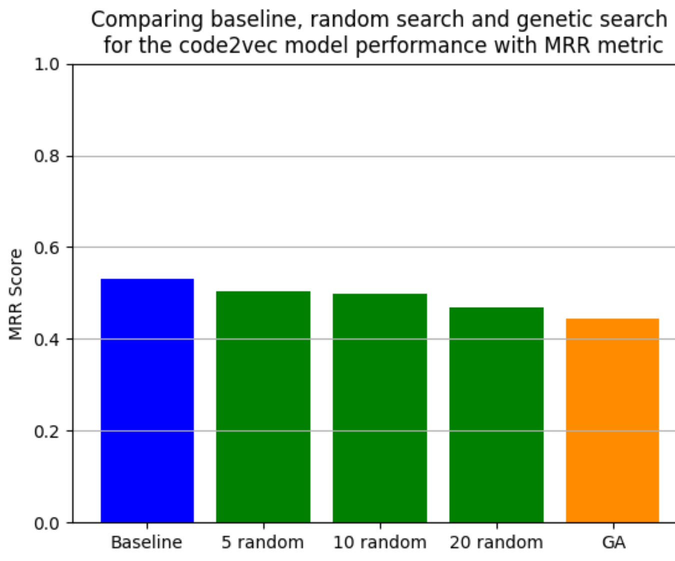


Figure A.8: Comparison baseline, random search, and genetic search with the MRR metric

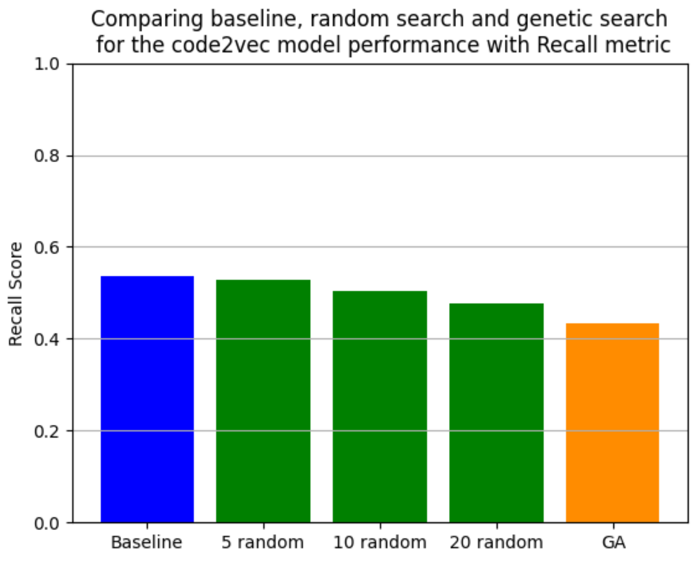


Figure A.9: Comparison baseline, random search, and genetic search with the recall metric

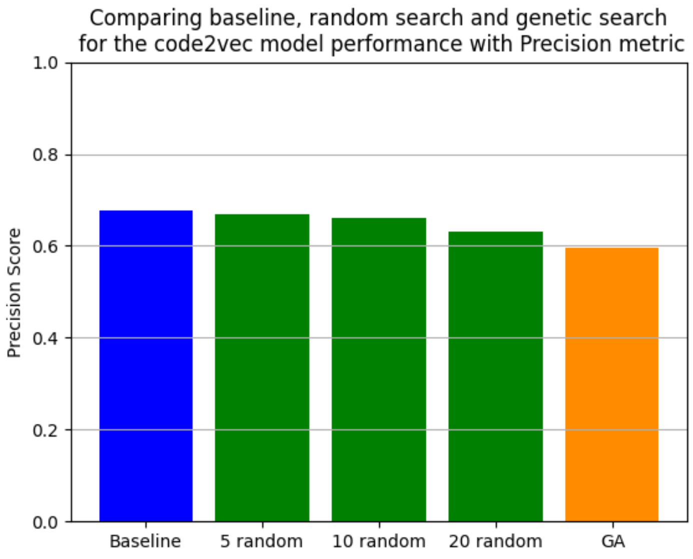


Figure A.10: Comparison baseline, random search, and genetic search with the precision metric

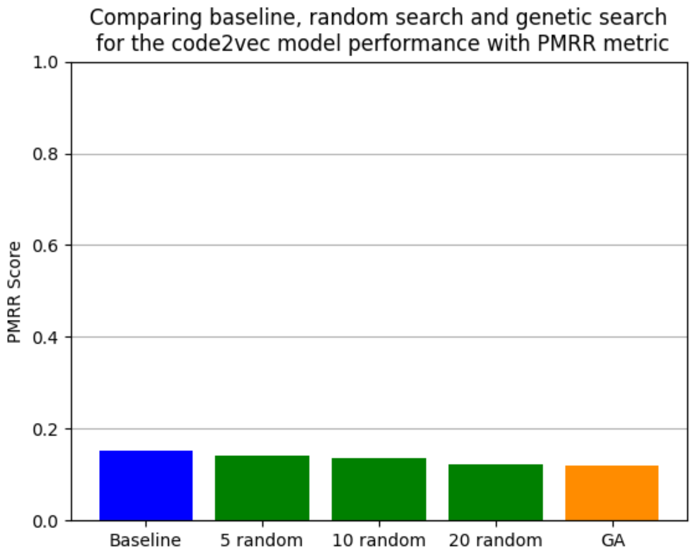


Figure A.11: Comparison baseline, random search, and genetic search with the percentage_MRR metric

A.3 Multi metric results

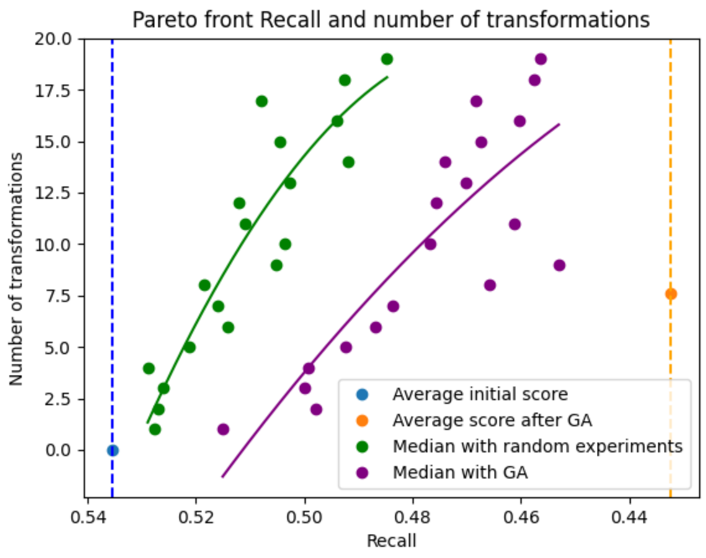


Figure A.12: Recall & number of transformations median Pareto

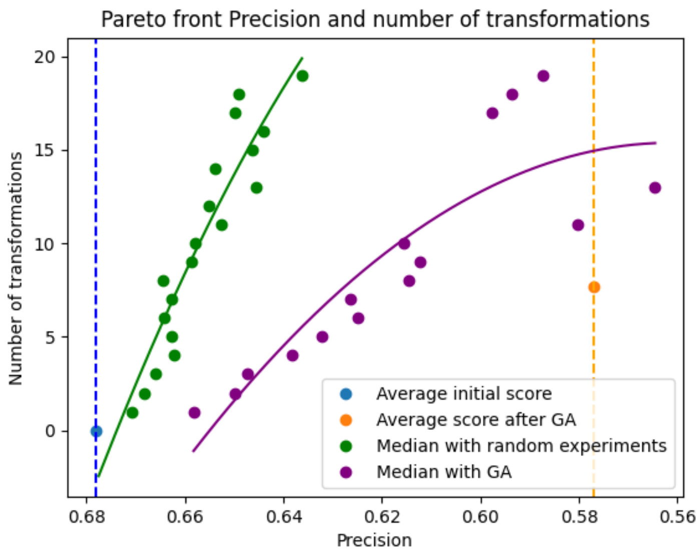


Figure A.13: Precision & number of transformations median Pareto

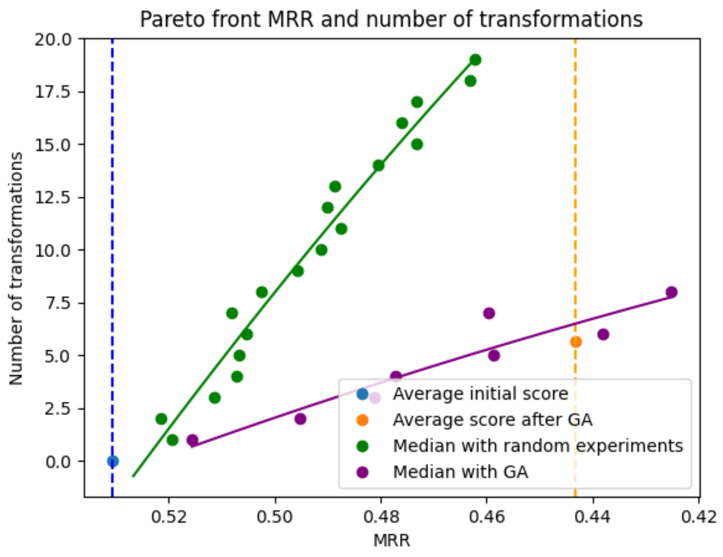


Figure A.14: MRR & number of transformations median Pareto

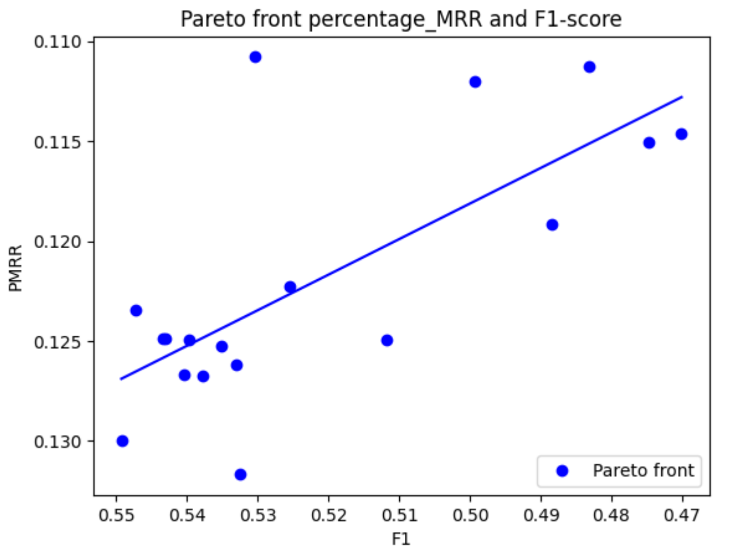


Figure A.15: Percentage_MRR and F_1-score metrics Pareto front measured with genetic search