Searching for Quality: Genetic Algorithms and Metamorphic Testing for Software Engineering ML

Applis, L.H.; Panichella, A.; Marang, R.J.

DOI
10.1145/3583131.3590379

Publication date
2023

Document Version
Final published version

Published in
GECCO 2023 - Proceedings of the 2023 Genetic and Evolutionary Computation Conference

Searching for Quality: Genetic Algorithms and Metamorphic Testing for Software Engineering ML

Leonhard Applis
L.H.Applis@tudelft.nl
Technical University Delft
Delft, The Netherlands

Ruben Marang
rmar@live.nl
Technical University Delft
Delft, The Netherlands

Annibale Panichella
A.Panichella@tudelft.nl
Technical University Delft
Delft, The Netherlands

ABSTRACT

More machine learning (ML) models are introduced to the field of Software Engineering (SE) and reached a stage of maturity to be considered for real-world use; But the real world is complex, and testing these models lacks often in explainability, feasibility and computational capacities. Existing research introduced metamorphic testing to gain additional insights and certainty about the model, by applying semantic-preserving changes to input-data while observing model-output. As this is currently done at random places, it can lead to potentially unrealistic datapoints and high computational costs. With this work, we introduce genetic search as an aid for metamorphic testing in SE ML. Exploiting the delta in output as a fitness function, the evolutionary intelligence optimizes the transformations to produce higher deltas with less changes. We perform a case study minimizing F1 and MRR for Code2Vec on a representative sample from `java-small` with both genetic and random search. Our results show that within the same amount of time, genetic search was able to achieve a decrease of 10% in F1 while random search produced 3% drop.

CCS CONCEPTS

• **Software and its engineering** → **Search-based software engineering**; **Software testing and debugging**; • **Computing methodologies** → *Neural networks.*

ACM Reference Format:
Leonhard Applis, Ruben Marang, and Annibale Panichella. 2023. Searching for Quality: Genetic Algorithms and Metamorphic Testing for Software Engineering ML. In *Genetic and Evolutionary Computation Conference (GECCO '23), July 15–19, 2023, Lisbon, Portugal.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3583131.3590379

1 INTRODUCTION

Producing a good model is hard. Not only is achieving good metrics often quite a challenge itself, but once entering the *real world* new problems emerge: Fairness [1], extrapolation [2], speed [3], security and robustness [4] are important non-functional requirements when it comes to machine learning (ML). And while good metrics make it into academic publications, poor non-functional qualities make the news [5]. In the realm of programming languages, ML applications have unique benefits, such as a large body of data publicly available in version control systems like GitHub or GitLab and discussion forums such as StackOverflow.

Unlike other domains, SE tasks have well-defined problems (e.g., code completion, test generation) and metrics (e.g., code coverage) that can be tackled with ML. Still, previous research shows that even in *clean* domains like software engineering problems with robustness and performance exist [6–9]. How is that?

We argue that one missing piece is the lack of tools for expressing non-functional requirements as actionable tests. A meta-survey on Requirements-Engineering for ML [10] found that a considerable amount of publications are aware of non-functional requirements, but few go beyond defining the problem. Hence, we have a rich concept of requirements, but any good requirement must be expressed as a (repeatable) test.

Within this work, we target **robustness** of ML models as an exemplary non-functional requirement. Robustness expresses the ability of the model to perform reliably when facing noise and degrading data quality. Such noise in code consists of poor naming-standards, unused elements and redundant structures. As the rules for programming languages are well-defined, we can produce noise while keeping the program identical in behavior using metamorphic transformations. Metamorphic transformations utilize metamorphic relations to generate new alternative datapoints for which a ML model *should* give the same prediction/classification outcome. A *robust* model is capable of detecting redundant elements and stay mostly unaffected by variable names. Robust models do not come for free and existing research shows that Code2Vec[11] is affected by metamorphic transformations; even just renaming variables can completely change its outcome [6].

Assessing robustness as part of quality is the job of a tester, adapted for working on machine-generated models instead of code written by human developers. We argue that a non-functional requirement like robustness can be expressed using a statistical test which is explainable in layman terms. Similar approaches have been done by previous research [7–9], which are limited by *blind* (random or stacking) application of transformations. To implement the test, we use a search technique (genetic algorithms) in combination with metamorphic transformations. Introducing evolutionary intelligence ought to deal with these two crucial limitations, realism and computational efficiency. We aim to produce datapoints creating similar deltas, while requiring less computation and enabling more *realistic* datapoints. The created datapoints can be saved and re-used, to re-evaluate the model, forming an acceptance test.

The contributions of this paper can be summarized as follows:

Gecco 2023, 15-19 July, 2023, Lisbon, Portugal

ACM ISBN 979-8-4007-0119-1/23/07…$15.00
https://doi.org/10.1145/3583131.3590379

(1) Formulation of metamorphic testing as a search based problem within the SE4ML domain
(2) Expand existing metamorphic testing within the domain with genetic search
(3) Assess differences between random and genetic search w.r.t. generating effective metamorphic tests that affect code2vec
(4) Sound statistical analysis based on multiple experiments utilizing genetic search, in particular for repeated transformations

Our results show that genetic search performs significantly better in reducing F1 score than random application of transformers. Based on the assumption that less transformations yield more realistic input-data, genetic search produces the same statistical difference in metrics with less transformations which we consider an increase in realism. Aligning with existing research, applying more transformation leads to bigger differences, which is further amplified if *the right* transformations are kept through genetic selection. An inspection of the most dominant transformation showed that especially those which add elements to the abstract syntax tree (AST) prevail, which we argue is connected to Code2Vecs' mechanisms based on AST-traversal.

2 BACKGROUND & RELATED WORK

2.1 Code2Vec & Method-name Prediction

Our experiments reuse the approach and artifacts[1] of Code2Vec [11] by Alon et al. Code2Vec is based on AST-path extraction for which code is translated into an AST and sampled into triples of `[leaf,path,leaf]`. These triples are merged into an vector-embedding per method, which shows promising results for the task of method-name prediction, and in particular their embeddings behave similar to generic nlp-embeddings. An important detail for this work is that the paths are extracted by performing random walks (default 200), limited by some further constraints (e.g. maximum width and depth). Due to the walks, the structure of the AST has paramount effect on the embeddings, as a single new leaf node doubles the amount of possible paths. Not only can the introduction of new nodes drastically change the amount of available paths, but also changes in the AST can lead to exclusion of previous valid paths. Hence, manipulation of the AST can lead to significant changes for better or worse - both important information as well as noise can be left out either by exclusion criteria or by chance.

Method-name prediction is a research area [12] where for a given method-body, a descriptive method-name is wanted. Descriptiveness is measured as F1-score by calculating sub-word token overlap of produced and actual method-names. Code2Vec outputs method-names with corresponding certainties, suitable to evaluate mean-reciprocal rank (MRR) that is based on the rank at which the correct prediction was placed. Within Code2Vec, the MRR was reported but only the F1-score was used for model training.

In addition to their work, Alon et al. provide java-datasets and we use *java-small* for our experiments. We sample 350 of the ~6000 files, which satisfies 95% significance at 5% error rate.

2.2 Metamorphic Testing

Metamorphic testing is based on the concept of *metamorphic relations*, relations that generate new test inputs that should result in the same test outcome. Humans as well as tools can easily create new test cases based on these relations. An overview of the metamorphic testing landscape can be found in a survey by Segura et al. [13]. While metamorphic testing is not yet widely adopted to machine learning for software engineering (**ML4SE**), both transformations and relations are known in SE and are well explored for *test case generation*, *refactoring*, program *optimization*, and linting.

Metamorphic Testing for ML. Metamorphic testing has gained popularity in machine learning, particularly in image-based object-detection tasks [14][15]. These transformations on images apply *information-preserving* changes to images: The picture of a dog can be mirrored, but a model should still be able to classify it as such.

Researchers have adapted and evaluated metamorphic testing to ML models in the ML4SE domain, i.e., to models that aim to semi-automate SE tasks. Compton et al. [7] introduced obfuscation techniques for variable names and showed how `Code2Vec` models are vulnerable to variable name changes. Our underlying framework and approach share similarities, which we extend by adding search algorithms. The combination of existing research (transformers for models of code), search algorithms and statistical tests forms the unique novelty of this paper.

Yefet et al. [6] proposed a further metamorphic relation introducing unused variables in addition to variable-name obfuscation. Their study on `Code2Vec`-based classifiers showed how these simple code-snipped transformations could generate adversarial attacks that fool the model under test. While this forms a search using metamorphic transformations, our approach differs in three primary aspects: ① we use Code2Vec as a black-box model, ② we approach search as a quantitative task on multiple (many) datapoints, instead of creating single counter-examples and lastly ③ we target robustness as a quality attribute, and not security.

Cito et al. [16] generated "*counterfactual examples*" to assess the robustness of BERT-like models. Although they use different terminology, these examples are generated by applying perturbations to initial code-snippets by replacing tokens with plausible alternatives that do not alter the code's behavior; hence, these transformations are metamorphic. Their study showed how transformations could find counterexamples for BERT-like models that are in line with the rationale provided by human experts.

A more extensive list of metamorphic transformations for ML-models applied to source code has been introduced in *Lampion* by Applis et al. [9]. In particular, Lampion considers multiple different metamorphic transformations, which either add unused information (e.g., add unused variables, input parameters, and wrap an expression in identity-lambda functions) or replace a code element with another equivalent element (e.g., rename a class, method or variable). Their study investigated the extent to which different transformations affect the performance of CodeBERT [17].

Limitations. Despite these undisputed advances, metamorphic tests for ML models are created in ML4SE by using random sampling. In particular, given a set of possible transformations, existing approaches randomly select and apply these transformations (one

[1] Note: We use a fork with minor changes due to machine dependencies https://github.com/ciselab/code2vec

or more times) until the model under test produces different test results (e.g., wrong classification). In this paper, we proposed the use of evolutionary intelligence (and evolutionary algorithms in particular) with the goal of ① leading to model miss-prediction faster via intelligent search and ② reducing the number of transformations needed to do so.

2.3 Genetic Algorithms

Search-based software testing (SBST) relies on search algorithms to seek for solutions to software testing problems. Since the 1990s, researchers have proposed and applied different meta-heuristics to optimize various testing problems, such as test case generation [18–20], regression testing [21, 22], and mutation testing [23]. The most applied meta-heuristics in the SBST literature include but are not limited to hill climbing [18], simulated annealing [24], and genetic algorithms [20, 25, 26]. Previous work has also shown how evolutionary intelligence can outperform random sampling (or random search) in specific testing applications [27–30], thus motivating our idea to use evolutionary testing for metamorphic testing.

Genetic algorithms (**GAs**) [31] are a group of search techniques inspired by *natural selection* and *natural evolution*. GAs evolve a pool of solutions (referred to as "individuals" or "chromosomes") or "population". Usually, the initial population is randomly generated and it is iteratively recombined and mutated using *crossover* and *mutation* operators. Solutions are selected for reproduction according to a fitness function that measures how good the solutions are toward solving a specific problem (e.g., generating tests that maximize coverage). Over the course of different iterations (or *generations*), this procedure of selecting, recombining, and mutating solution converges towards best-fit solutions. GAs terminate when either the optimal solution to the problem is found or when the search budget (e.g., the number of generation) is depleted. For a detailed reading on the matter, we suggest the work by Sette et al. [32].

3 APPROACH

Proofing a program (or here a model) to be correct is generally unfeasible; instead, one tests for failures. Programs are asserted for a general quality expressed through *happy-paths*, and checked for negative behavior, error recovery, and other issues through tests. In this tradition, we also design our model-test-cases failure-based: We are looking for input that makes the model perform poorly. We assume the *happy-path* is successfully covered through the performance in training and test.

Creating a single faulty data point to produce errors is an easy task, but also one that does the model's statistical nature injustice: The issue here is that single data points likely will not be useful in producing a fix. An easy way for a model to deal with a single faulty data point would be over-fitting to avoid this particular data point, and any generalization could be mere coincidence. Hence, we zoom out a bit and consider approaches that focus on multiple data points and attributes of the dataset.

How would you go about finding these datapoints? We formulate this as a classic **search problem**:

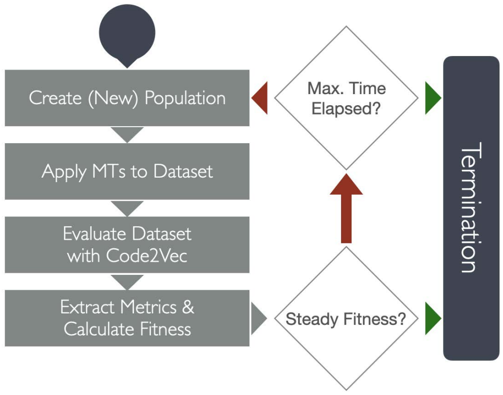

Figure 1: schematic Control-Flow of Guided-MT Program

Definition 1. *Let $X : P \longrightarrow O$ be a pre-trained model that takes as input a program P and returns an outcome $o \in O$ (e.g., method-name prediction). Let $F = \{f_1, \cdots, f_n\}$ be a set of metamorphic transformations such that $f_i(P) \equiv P \;\; \forall f_i \in F$. Let $m : (X, P) \longrightarrow \mathbb{R}$ be a performance metric (e.g., F1-score) computed on a pre-trained model X with input program P.*
Problem*: finding a program P' obtained by applying F to P that maximizes the differences in the performance metric m:*

$$\max \left| m(X, P') - m(X, P) \right| \text{ with } P' \equiv P \tag{1}$$

The formulation above can be applied to any performance metric. In this paper we focus on F1-score; our goal consists in finding a program P' that is equivalent to an initial program P (hereafter referred to as *seed*) that maximizes the difference in F1-score achieved by a model X on P and P', i.e., $\max |F1(X, P') - F1(X, P)|$. Equation (1) corresponds to our fitness function to optimize for metamorphic testing (our search guidance).

3.1 Guided Metamorphic Testing

To optimize Equation (1), we implemented a genetic algorithm, whose high-level workflow and its components are depicted in Figure 1. Within our experiments, we focused on Java programs and target `Code2Vec` as main the model under test. For `Code2Vec`, we re-use the artifacts provided by Alon et al. [11], including code, model, and dataset. To apply the metamorphic transformations, we use the Lampion framework [9] since it provides the most extensive set of metamorphic transformations for Java programs.

3.1.1 Encoding. As mentioned before, solutions to the problem in Definition 1 are produced by altering a *seed* program P by applying metamorphic transformations. Instead of encoding a solution as a complete code-snippet, we only encode the changes applied to the AST of the seed image (*mask*). In particular, given the seed program P, we encode a solution (*genotype*) as a sequence of changes to the AST of P: $P' = \langle p_1, \ldots, p_k \rangle$. Each entry p in P' is a tuple $[node_i, f_j]$, where $node_i$ denotes the i-th nodes in the AST of P and f_j is the j-th metamorphic transformation applied to that AST node.

3.1.2 Initialization. The first step for our GA requires creating an initial population of metamorphic tests (or solutions). The initial population consists of creating N copies of the seed program P (i.e.,

Table 1: List of metamorphic transformations (MTs) for Java programs [9].

ID	Description
MT-IF	Wrapping a random expression in an if(true) statement
MT-FI	Wrapping a random expression in an if(false) else statement
MT-UV	Add a random unused variable
MT-RE	Rename a variable
MT-PR	Rename a parameter
MT-ID	Wrap an expression in an identity-lambda function (including function call)
MT-NE	Add the neutral element to a primitively typed expression

empty mask $S = \langle\rangle$) and randomly applying one of the available metamorphic transformations at a randomly selected AST node of P. In this paper, we consider seven metamorphic transformations proposed in Lampion [9] and listed in Table 1.

3.1.3 Selection. Metamorphic tests are selected for reproduction using simple *tournament selection* with a tournament size ts =4 [32]. This selection method first randomly samples four solutions from the last population and selects the solution with the best fitness function (Equation (1) in our case) as the winner of the tournament (parent).

3.1.4 Crossover. New solutions/tests are generated by recombining parent solutions selected as described in the previous subsection. In particular, we apply the multi-point (also known as scattered) crossover. This operator creates two new metamorphic tests by combining the entries of two parent solutions around multiple cut points. First we create a *crossover-mask* consisting of a binary vector with randomly generated entries, this entry determines whether a crossover at this position will take place. Each offspring is created as a copy of one of the two parents, and the transformation at index i is replaced by the gene of the other parent if the mask entry at i is $true$.

The masks' length is bounded by the shortest gene, in case of diverging length the crossover happens between the overlapping genes and the remaining genes are left untouched in the longer offspring (copy of the longer parent).

3.1.5 Mutation. Given a newly generated test O, we designed two types of mutation operators that in turns *add* or *delete* metamorphic transformations to O. The probability of application is configurable (for the experiments we chose 80% add and 20% shrink), the application is mutually exclusive. The chance to trigger a mutation is set to 50%, leading to (in average) one of the offsprings to be mutated.

The *add* operator iteratively adds multiple metamorphic transformations following a hyperbolic distribution. In detail it adds a (randomly selected) metamorphic transformation to O to an AST node with probability σ. Once it is added, a second (randomly selected) transformation is applied with probability σ^2, and so on until no more transformation is added. In general, at each mutation iteration n, a new transformation is added with probability σ^n. Notice that a new transformation is added if and only if the limit $L = 20$ is not reached. This non-linear operator is inspired by the *add* operator used in EvoSuite [19, 33] for unit-test suite/-case generation. In this paper, we set $\sigma = 2/3$ as it asymptotically applies three metamorphic transformations on average (statistical *expectation*).

The *remove* operator randomly removes one metamorphic transformation previous applied to O. This corresponds to deleting one of the entries o_j in the mask $O = \langle o_1, \ldots, o_k \rangle$, with $j \in \{1 \ldots k\}$. This operator tackles the potential *bloating effect* [34, 35], i.e., the length of the metamorphic tests might steadily increase through the generations. This *remove* operator can shorten solutions with spurious transformations that do not contribute to the fitness function throughout the generations.

3.1.6 Elitism. At the end of each generation, there are N parent solutions and N offspring solutions produced via the selection, crossover, and mutation. The new population for the next generation is obtained by selecting the N best solutions among parents and offspring. This survival mechanism is called *elitism* since the best solutions can survive across the generations.

3.1.7 Termination. The search terminates when the total search budget is reached or the fitness function cannot be further improved. We prefer running time over number of iterations as a search budget as it is considered the fairest metric to measure the cost of test generation approaches [27, 28, 30]. This is because the cost of applying the genetic operators (<10 seconds) is —in our context— negligible compared to the cost of computing the fitness function, which requires evaluating the generated solutions against the model under test (2-10 minutes for Code2Vec).

4 METHODOLOGY

4.1 Research Questions

Based on existing research (Section 2) Code2Vec is vulnerable to metamorphic transformations, especially centered around identifiers. Thus, we want to investigate if genetic search improves the produced effect and the speed needed to influence the model. Differences between *random search* and *evolutionary search* for a performance drop are covered in the first research question:

RQ1: Effectiveness of Search

How effective is searching for metamorphic transformations that produce a maximum drop in performance metrics (F1, MRR) of Code2Vec?

The random search adds a number of randomly chosen transformations. Primarily it is the search technique used in the existing body of research w.r.t. metamorphic testing in the ML4SE domain [6, 9, 16] and it is the current state-of-the-art. Second, random search is natural baselines when assessing search-based approaches considering its simplicity and strength. Previous studies showed that random search can outperform evolutionary algorithms in specific SE domains [36, 37].

We want to explore trade-offs between the number of transformations and their produced effect. We expect a larger number of transformations to produce bigger changes in Code2Vec output, but how much difference can be achieved for fixed n transformations? A low number of transformations keeps the code understandable, i.e., a few redundant control structures or variable names could very well be an oversight in *normal* programming. These realistic data points are the golden fleece of our second research question:

RQ2: Minimizing Number of Transformations

What are the trade-offs between keeping a low amount of transformations and producing a difference in metrics?

To position this work better in the body of existing research, we unravel the genotypes and investigate the obtained transformations. With our third research question, we aim to get an insight into the models behavior and/or data — we aim to answer a set of questions such as: *"Align our distributions of transformers with other publications?"* or *"For different metrics, do we get different transformations?"*

RQ3: Distribution of Transformations

What are the dominant transformation that leads to highest changes in performance?

Existing work shows that metamorphic transformations reduce the metrics *on average* — but individual transformed datapoints can produce a better prediction. We consider this an exotic premise: Can noise make our data better? We hope to gain insights on the topics data quality, model fitting and the stochastic nature of ML with our last research question:

RQ4: Search-Goal Inversion

Can the search be inverted, to produce an increase in performance metrics?

4.2 Benchmark and Dataset

`Code2Vec`, the model under test, is a neural attention-based approach that learns embeddings for programs (e.g., method-names) as continuous distributed vectors. The code embedding aims to preserve the semantics of the programs such that semantically similar methods are mapped into similar vectors. To this aim, programs are represented as path contexts, i.e., paths between nodes on the program AST.

We use the pre-trained model by Alon et al. [11], trained on more than 12M Java methods extracted from 10,072 Java projects available on GitHub. Together with the pre-trained model, the training-, validation-, and test-sets are also available in the replication package by Alon et al. [11].

For our paper, we focus on the pre-trained model and use the `java-small` test-set, which contains ~6000 Java methods. We randomly sampled 350 methods from the test-set to use a representative sample. We restricted our evaluation to a smaller set of methods in the test-set because of the large number of runs needed for a sound statistical analysis. We had to choose between using a larger sample of seed programs with few runs or a smaller sample but with enough runs to allow sound statistical analysis within a feasible wall time. We have opted for the latter based on the existing guidelines on assessing randomized algorithms in SE [38, 39], which highlight the importance of performing multiple runs for a proper assessment of search algorithms (including random search and genetic algorithms).

4.3 Evaluation Methods

In our experiment, we run GA and random search 10 times for each program seed in the dataset; this accounts for the random nature of the employed search algorithms. In each run, we collected the best solution (i.e., the metamorphic tests with the best fitness function value), its corresponding performance metrics, and its sequence of metamorphic transformations such solution included. We reran the same experiments twice, once for each fitness function in Equation (1) instantiated with F1-score and a second time using MRR (Mean Reciprocal Rank). In total, for RQ1, we run 350 (seed programs) × 2 (algorithms) × 10 (runs) × 2 (fitness functions) = 14,000 runs.

To answer RQ1, we compare the average (median) differences in F1-score and MRR scores achieved by the two search algorithms in the comparison. Note that achieving a lower F1-score indicates a better ability of a search algorithm to find metamorphic tests that impact `Code2Vec`. In addition to analyzing the median results, we also applied sound statistical tests as suggested by Arcuri and Briand [38, 39].

In particular, we applied the Wilcoxon rank sum test [40], with threshold p-value=0.05. We complement the Wilcoxon test with Vargha-Delaney $\hat{A}_{12}$ statistics [41], which provides a measure of the effect size (or magnitude of the difference). $\hat{A}_{12}$=0.50 indicates that two distributions in the comparison (e.g., F1-score drops achieved by GA and random search) are equivalent. For F1-score, $\hat{A}_{12} > 0.50$ indicates that GA achieves a significantly larger drop in F1-score compared to random search (i.e., GA is better). $\hat{A}_{12}$ also provides an easy-to-interpret classification of the effect size in *negligible*, *small*, *medium*, and *large* [41].

We use non-parametric tests for both significance and effect size over parametric alternatives (e.g., the paired t-test) because the data does not follow a normal distribution [42][2].

To answer RQ2, we compare the number of transformations required by genetic algorithms and random search to achieve the same delta for `Code2Vec`. In particular, we analyze how F1-score and MRR vary when applying varying number of metamorphic transformations.

For RQ3, we performed a deeper analysis of the data collected for RQ1. We analyze the genotype (sequence of applied transformation) of the best solution produced by random search and GA in each individual run. Our goal is to determine whether certain metamorphic transformations appear more than others in the best solutions.

Finally, we re-run the experiment for RQ1 but search for improving rather than decreasing the performance metrics for *Code2Vec*, i.e., increasing the F1-score and reducing MRR. We want to understand the extent metamorphic testing could be used to increase the robustness of *Code2Vec* rather than looking for adversary examples in which it is vulnerable. Hereafter we refer to F1-min and F1-max to distinguish between the setting used in RQ1 to assess the robustness of `Code2Vec` and the one used in RQ4 to strengthen the performance of `Code2Vec`.

4.4 Experiment Setup

We ran the experiments utilizing CPUs on a server with an AMD EPYC 7H12 64-Core Processor. We conducted ten experiments in parallel, which lead to a total wall-time of 3 days and a computation-time of ~400h. We provide all experiments, data and model within

[2] We pre-tested the nature of the data distribution using the Shapiro-Wilk test of normality [43]

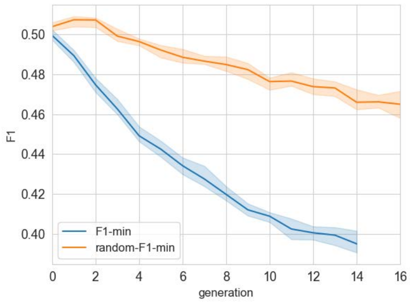

Figure 2: Comparison of F1 for random and genetic search

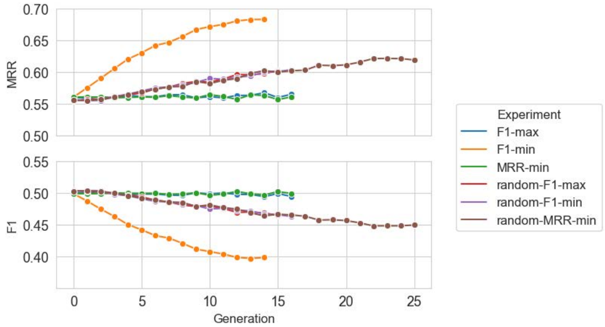

Figure 3: Overview of Metric-Movements

a replication package https://doi.org/10.5281/zenodo.7306931 . The code is separately provided at https://doi.org/10.5281/zenodo.7307012 and results are available at https://doi.org/10.5281/zenodo.7307208 .

For the GA, we set a small population size of 10 individuals. This choice considered the high cost of the fitness evaluation (against the model), which can take between 2 and 10 minutes for our population size. Small population size is widely recommended in the literature for expensive fitness functions [44, 45]. For the selection operator, we used tournament selection with a tournament size of 4, which allows better exploitation and fast convergence rate [32]. The mutation rate is set to 0.50, which is relatively high, but it prevents genetic drift in case of a small population size [32]. Finally, we set the crossover rate *cr*=1.00, which is within the recommended range [46]. For both random search and genetic algorithm, we use the same termination criteria of 360 minutes search time. For GA, we terminate earlier if the (best) fitness has not changed for 8 continuous steady generations. Most of the experiments terminated around 4 hours due to convergence.

5 RESULTS

5.1 RQ1 - Effectiveness of Search

Figure 3 shows an overview of the achieved changes in metrics for different experiment setups, with a detailed view on the primary experiments in Figure 2. Experiments prefixed with *random* exploit random search, while those without utilize genetic search. Most of the configurations do not achieve a visible difference of metrics within the search budget, except for `F1-min` detailed in Figure 2. Table 2 summarizes the statistical tests for comparison of `F1-min` and *random-F1-min*: Both experiments achieve a statistical significant difference and `F1-min` achieves higher levels of difference quicker than its random pardon. On average random search needs three generations for 1% drop in F1 while genetic search needs two. The reported Wilcoxon p-value prooves that there is a significant different distribution for the algorithms, and effect size ($\hat{A}_{12}$) shows that after one generation there is a large difference between random and genetic search.

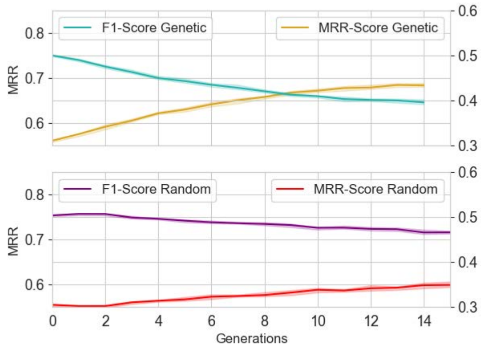

Figure 4: Metric-movement for random & genetic of F1-min

The second biggest difference in F1 are achieved by `random-MRR-max/min`, which *simply* apply random transformations. Random search produces less movement than `F1-min`, but creates a higher delta than genetic search for MRR or `F1-max`.

Our setup seems unable to search for a change in MRR (at least as an dedicated objective). We expect this to be inherent for the model as it was trained solely for F1 and MRR was only reported for comparison with other research. Hence, while side-effects are possible, we consider that the model is *blind* towards unknown metrics — which is somewhat expected. Within the `F1-min` experiment we see an unexpected effect: While the F1-score is dropping, the MRR rises accordingly.

The counter-play of F1 and MRR seems to be related with the dataset and the properties of the metrics. In general, the method-names of the datasets are short and most are between 2-4 sub-tokens. The average prediction without any MTs tends to be slightly longer (2-6 tokens). Minimizing F1 pushes the predictions to be shorter, which as a side-effect moves the predictions more towards the right token-distribution, achieving a better MRR in the process. The shorter words are *worse* for F1, as in general longer words are more forgiving. Given 5 or 6 sub-tokens, a partial overlap can produce some scores, but with 1 or 2 sub-tokens it is *"hit or miss"*.

Summary RQ1

Within 15 generations, we found datapoints resulting in a drop of up to 10% in F1-score. While the F1-Score drops for this experiment, the MRR rises respectively. Random search performs about half as good the genetic search, but is in itself significant.

Table 2: Statistical Tests for F1 Score and MRR

Gen	Results for F1-score					Results for MRR				
	Random	GA	p-value	$\hat{A}_{12}$		Random	GA	p-value	$\hat{A}_{12}$	
0	0.50	0.50	<0.01	0.59	small	0.56	0.56	<0.01	0.41	small
1	0.51	0.49	<0.01	0.84	large	0.55	0.57	<0.01	0.17	large
2	0.51	0.47	<0.01	0.93	large	0.55	0.59	<0.01	0.07	large
3	0.50	0.46	<0.01	0.91	large	0.56	0.60	<0.01	0.07	large
4	0.50	0.45	<0.01	0.97	large	0.56	0.62	<0.01	0.03	large
5	0.49	0.44	<0.01	0.97	large	0.57	0.63	<0.01	0.02	large
6	0.49	0.43	<0.01	0.97	large	0.57	0.64	<0.01	0.03	large
7	0.49	0.43	<0.01	0.97	large	0.57	0.65	<0.01	0.02	large
8	0.48	0.42	<0.01	0.98	large	0.58	0.66	<0.01	0.01	large
9	0.48	0.41	<0.01	0.99	large	0.58	0.67	<0.01	0.01	large
10	0.48	0.41	<0.01	0.99	large	0.59	0.67	<0.01	0.01	large
11	0.48	0.40	<0.01	0.98	large	0.59	0.68	<0.01	0.02	large
12	0.47	0.40	<0.01	0.99	large	0.59	0.68	<0.01	0.01	large
13	0.47	0.40	<0.01	0.99	large	0.59	0.68	<0.01	0.01	large
14	0.47	0.40	<0.01	0.99	large	0.60	0.68	<0.01	0.00	large

5.2 RQ2: Minimizing Number of Transformations

In terms of (co-)relations between transformations and deltas in metrics, the clear trend we found was a simple correlation between produced movement and number of transformations: More transformations produce a higher change, being nearly proportional (See Table 2 and Figure 2). Over the generations, initial iterations produced a higher drop in metrics and more transformations being added which eased out in later generations, however the symmetry between transformations and deltas persists.

Our intended tradeoff-analysis utilizing a weighted-sum approach failed due to this correlation: When equally weighting number of transformations and the observed delta, the fitness remained at roughly the same value producing a stale search, degrading into behavior similar to random search. Introducing more sophisticated approaches such as multi-objective optimization over different metrics is considered valuable future work, as we attribute issues solely to a simplistic fitness function.

Summary RQ2

Movements in metrics are proportional with the amount of transformations. A weighted-sum approach to find tradeoffs failed due to this correlation.

5.3 RQ3: Distribution of Transformations

Figure 5 shows the distribution of transformers over generations for minimizing the F1-score. Over the generations, it crystallizes that IfTrue Transformations and Lambda-Identity-Transformations seem to have the greatest effect on metrics, while renaming variables occur the least.

We attribute this to the embedding-logic as presented in section 2.1 as the prominent transformations alter the AST quite heavily: The added redundant condition adds a total of 6 nodes to the AST, the lambda creates additional 5. On the contrary, renaming a variable adds no node and are less represented in our results. The work by Compton et al. [7] proves that the variable names play an important role in prediction, but given our results, it seems that structural changes out-weight the changes in information, i.e. the form of the AST weights more than the content of the nodes.

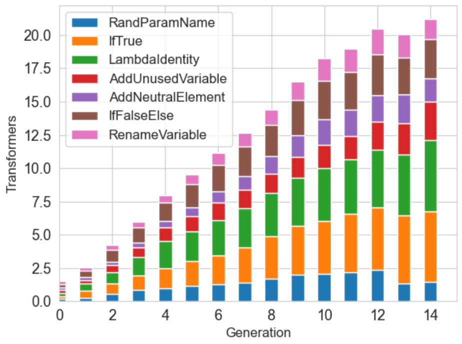

Figure 5: Applied Transformers for minimizing F1-score

The *failed* experiments (MRR-based and F1-maximization) show a near-evenly distributed composition of transformers. They behave parallel to random search, and form another piece of evidence that with the current model and approach we cannot search for these optimization goals.

Summary RQ3

The most common transformations were `IfTrue` and `LambdaIdentity`. Transformations seen in existing research had less impact than these structural changes.

5.4 RQ4: Search-Goal Inversion

Initially this RQ was inspired by existing research [7, 9] that found *flaky* datapoints when applying transformations: some got worse, while others got better, with the average being worse. We expected to find symmetrical behavior for maximizing and minimizing alike;

If we can find datapoints that produce worse metrics once we add noise, we found a model that overfits on *clean data*. If we can find datapoints that produce better metrics once we add noise, we found a model that is still underfit.

Our experiments show that with the presented approach we cannot search for a maximization, or that the model is truly robust against the changes. Regarding the latter, we still observe the flakiness, but there is no clear *movement on average*. Further generations produce stronger oscillating results, but it cannot keep positive-changes while discarding those that decrease, simply because any change applies in both directions.

It is however strange that the approach did not *proxy* the search for MRR by the inverse optimization of F1 — after all, given Figure 3 they seem near correlated.

Summary RQ4

Neither maximizing F1 nor MRR was possible. The MRR noticeably increased when minimizing F1, but this is related to a specific attribute in F1 for the dataset.

6 DISCUSSION

MRR Experiments. It is puzzling that searching for MRR did not succeed, despite `F1-min` quite successfully maximizing MRR. Why does the `MRR-max` not simply do what `F1-min` does? Looking back at the setup, the error seems to be in the `Genetic Search`. Measurement and evaluation works, and creation of datapoints that increase MRR is successful as per `F1-min`. The reasons could either be related to the feedback-loop as a whole, or be inherent to the search algorithm and its configuration.

To solve this open question, we suggest further research including a model trained on both metrics. We envision a set of models trained for MRR, F1 and in best case both, adopting the experiment from this work for each model. If we observe the same blindness towards F1 when trained on MRR, we see a connection between training metrics and search success.

Failed Maximizing. The performed experiments failed to achieve a maximization of metrics showing a behaviour similar to random search. With the current configuration we apply a change to every datapoint in the test-set, but a more fine grained application is possible. In theory, a gene could be constructed consisting of `changes-per-file`. While this forms classic *future research*, we want to take a moment to dis-encourage attempts: The used dataset with 680 methods forms a representative sample of the original dataset, with an average F1-score of 0.50 Gaussian distributed. Even if every single transformation would maximize F1 for a given datapoint from 0 to 1, this transformation will at most contribute $\frac{1}{680} = 0.14\%$ towards a better F1 score. To achieve movements similar to those observed in this work, hundreds of generations are necessary[3].

7 THREATS TO VALIDITY

Construct validity While we assume the metamorphic tests generated by either GA or random search are equivalent to the original seed programs (due to the metamorphic relations), the resulting mutated programs might not be realistic (e.g., too many nested if conditions).

Internal validity. We selected 350 programs from the `Code2Vec` test-set using a randomised sampling that ensured diverse programs were selected considering (1) the original source project from GitHub, (2) the application domain, and (3) code characteristics (e.g., code complexity). We picked a representative sample size; however, due to implementation details, we had to re-draw the sample in some corner cases. Some elements were unsupported by our transformers, such as java files consisting of (only) enums. The final elements are unaltered from the original dataset and are provided in the replication package. In a similar direction, some files contained multiple classes, and many of the files contained varying amounts of methods — leading to different *weight* in the fitness calculation as we apply the transformers *per class*. We consider these uncertainties to be addressed by our statistically significant sample size.

Conclusion validity. To address the randomness in the search process, we ran each search algorithm ten times on each seed program with a different random seed (for the random number generator) in each run. Besides, we applied statistical tests (i.e., the Wilcoxon rank sum test and the Vargha-Delaney statistics) following the existing guidelines on how to assess randomized algorithms [38, 39]. While the choice to re-run the algorithms multiple times reduced the number of seed programs we could consider, our analysis is statistically sound.

8 CONCLUSION

The goal of this paper is to expand existing metamorphic testing for SE4ML with an evolutionary search to save on computational costs and produce more realistic data points (w.r.t. the number of applied transformations). To that end, we implemented a java program combining Code2Vec and a genetic algorithm using the models' metrics as fitness functions. We designed an experiment sampling a representative amount of data points from Code2Vecs' `java-small`-dataset and tried to minimize F1 and MRR with both random and genetic search.

Our results show that both random and genetic search significantly change the model metrics, with genetic search being stronger with progressing generations (determined per effect size) and leading to a total reduction of 10% in F1 for genetic and 3% in F1 for random. Minimizing MRR did not succeed with genetic search, performing similar to random search, likely due to the model being trained solely on F1 score. We found a near-proportional relation between change in metrics and applied transformations, failing our trade-off analysis experiments due to leveling the weighted-sum fitness function.

In summary, genetic search improved the existing research by proposing a more intelligent way to generate example. We consider our successes a worthwhile adoption for current approaches and our failures to be good starting points to address topics such as derived metrics and tradeoff analysis.

REFERENCES

[1] Z. Chen, J. M. Zhang, M. Hort, F. Sarro, and M. Harman, "Fairness testing: A comprehensive survey and analysis of trends," *arXiv preprint arXiv:2207.10223*, 2022.

[3] This is only for `java-small` — the other available datasets are magnitudes bigger

[2] G. Hooker, *Diagnostics and extrapolation in machine learning.* stanford university, 2004.

[3] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on tensorflow program bugs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 129–140.

[4] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," *IEEE Transactions on Software Engineering*, 2020.

[5] J. Zou and L. Schiebinger, "Ai can be sexist and racist—it's time to make it fair," 2018.

[6] N. Yefet, U. Alon, and E. Yahav, "Adversarial examples for models of code," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–30, 2020.

[7] R. Compton, E. Frank, P. Patros, and A. Koay, "Embedding java classes with code2vec: Improvements from variable obfuscation," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 243–253.

[8] M. R. I. Rabin, N. D. Bui, K. Wang, Y. Yu, L. Jiang, and M. A. Alipour, "On the generalizability of neural program models with respect to semantic-preserving program transformations," *Information and Software Technology*, vol. 135, p. 106552, 2021.

[9] L. Applis, A. Panichella, and A. van Deursen, "Assessing robustness of ml-based program analysis tools using metamorphic program transformations," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 2021, pp. 1377–1381.

[10] K. Ahmad, M. Bano, M. Abdelrazek, C. Arora, and J. Grundy, "What's up with requirements engineering for artificial intelligence systems?" in *2021 IEEE 29th International Requirements Engineering Conference (RE).* IEEE, 2021, pp. 1–12.

[11] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[12] K. Meinke and A. Bennaceur, "Machine learning for software engineering: Models, methods, and applications," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, 2018, pp. 548–549.

[13] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés, "A survey on metamorphic testing," *IEEE Transactions on software engineering*, vol. 42, no. 9, pp. 805–824, 2016.

[14] X. Xie, J. W. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen, "Testing and validating machine learning classifiers by metamorphic testing," *Journal of Systems and Software*, vol. 84, no. 4, pp. 544 – 558, 2011, the Ninth International Conference on Quality Software. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121210003213

[15] C. Murphy, G. Kaiser, L. Hu, and L. Wu, "Properties of machine learning applications for use in metamorphic testing," *20th International Conference on Software Engineering and Knowledge Engineering, SEKE 2008*, pp. 867–872, 2008.

[16] J. Cito, I. Dillig, V. Murali, and S. Chandra, "Counterfactual explanations for models of code," in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, 2022, pp. 125–134.

[17] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[18] P. McMinn, "Search-based software test data generation: a survey," *Software testing, Verification and reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[19] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 416–419.

[20] P. Tonella, "Evolutionary testing of classes," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 4, pp. 119–128, 2004.

[21] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software testing, verification and reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[22] ——, "Pareto efficient multi-objective test case selection," in *Proceedings of the 2007 international symposium on Software testing and analysis*, 2007, pp. 140–150.

[23] R. A. Silva, S. d. R. S. de Souza, and P. S. L. de Souza, "A systematic review on search based mutation testing," *Information and Software Technology*, vol. 81, pp. 19–35, 2017.

[24] H. Waeselynck, P. Thévenod-Fosse, and O. Abdellatif-Kaddour, "Simulated annealing applied to test generation: landscape characterization and stopping criteria," *Empirical Software Engineering*, vol. 12, no. 1, pp. 35–63, 2007.

[25] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Ieee transactions on software engineering*, vol. 38, no. 1, pp. 54–72, 2011.

[26] M. Martinez and M. Monperrus, "Astor: A program repair library for java," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 441–444.

[27] J. Campos, Y. Ge, N. Albunian, G. Fraser, M. Eler, and A. Arcuri, "An empirical evaluation of evolutionary algorithms for unit test suite generation," *Information and Software Technology*, vol. 104, pp. 207–235, 2018.

[28] A. Panichella, F. M. Kifetew, and P. Tonella, "A large scale empirical comparison of state-of-the-art search-based test case generators," *Information and Software Technology*, vol. 104, pp. 236–256, 2018.

[29] C. Birchler, S. Khatiri, P. Derakhshanfar, S. Panichella, and A. Panichella, "Single and multi-objective test cases prioritization for self-driving cars in virtual environments," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2022.

[30] S. Panichella, A. Gambi, F. Zampetti, and V. Riccio, "Sbst tool competition 2021," in *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST).* IEEE, 2021, pp. 20–27.

[31] J. H. Holland, *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* MIT press, 1992.

[32] S. Sette and L. Boullart, "Genetic programming: principles and applications," *Engineering applications of artificial intelligence*, vol. 14, no. 6, pp. 727–736, 2001.

[33] A. Panichella, F. M. Kifetew, and P. Tonella, "Reformulating branch coverage as a many-objective optimization problem," in *2015 IEEE 8th international conference on software testing, verification and validation (ICST).* IEEE, 2015, pp. 1–10.

[34] G. Fraser and A. Arcuri, "Evosuite: On the challenges of test case generation in the real world," in *2013 IEEE sixth international conference on software testing, verification and validation.* IEEE, 2013, pp. 362–369.

[35] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, 2017.

[36] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 254–265.

[37] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization." *Journal of machine learning research*, vol. 13, no. 2, 2012.

[38] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proceedings of the 33rd international conference on software engineering*, 2011, pp. 1–10.

[39] ——, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.

[40] W. J. Conover, *Practical nonparametric statistics.* john wiley & sons, 1999, vol. 350.

[41] A. Vargha and H. D. Delaney, "A critique and improvement of the CL common language effect size statistics of McGraw and Wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[42] M. D. Smucker, J. Allan, and B. 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.

[43] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," *Biometrika*, vol. 52, no. 3/4, pp. 591–611, 1965.

[44] R. B. Abdessalem, A. Panichella, S. Nejati, L. C. Briand, and T. Stifter, "Automated repair of feature interaction failures in automated driving systems," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 88–100.

[45] T. Chugh, K. Sindhya, J. Hakanen, and K. Miettinen, "A survey on handling computationally expensive multiobjective optimization problems with evolutionary algorithms," *Soft Computing*, vol. 23, 2019.

[46] H. G. Cobb and J. J. Grefenstette, "Genetic algorithms for tracking changing environments." Naval Research Lab Washington DC, Tech. Rep., 1993.