TU WIEN Informatics

Characterizing Counterfactual Explanation Search Spaces

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Christian Lutnik, BSc
Matrikelnummer 11808598

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dr.sc. Jürgen Cito, BSc

Wien, 31. August 2024

Christian Lutnik	Jürgen Cito

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Characterizing Counterfactual Explanation Search Spaces

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Christian Lutnik, BSc
Registration Number 11808598

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Ing. Dr.sc. Jürgen Cito, BSc

Vienna, August 31, 2024

Christian Lutnik	Jürgen Cito

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Christian Lutnik, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 31. August 2024

Christian Lutnik

Acknowledgements

From the very first days of my life, a path has been laid out before me that would eventually lead me to the completion of this thesis. And even with knowing as little of life as I know now, I am well aware that paving this path must not have been easy for those who did it for me. I can only begin to imagine the personal and professional sacrifices that both Mag. phil. Bernadette Lutnik and HR Dir. Dipl.-Ing. Hubert Lutnik have made and still do to remove obstacles and to ensure my progress on this path, thereby provide me with all the best education I could get. For this, I will be forever grateful to them and to the rest of my family.

To continue further in life, one also needs the support of friends, advisors, and teachers, which I was lucky enough to have enjoyed along my way. I am especially thankful for Simon Dörrer, MSc ETH QE and Dipl.-Ing. Marc Goritschnig, who provided all the help I needed to get through high school and university, and for Associate Prof. Dipl.-Ing. Dr.sc. Jürgen Cito who advised me on this thesis.

In his book "Vom Kriege", Carl von Clausewitz claims that morale is amongst the most important aspects of war. I find this claim to translate into the civilian world very well, where a great deal of it is required to complete any nontrivial task. During the most straining days my of studies, my morale has been greatly improved after meeting Chiara Egger, BSc, who gave me as much, if not more, emotional support as others have given me professional support.

Even though the list would be too long to name all of them here, I thank everyone who went this path along with me, who made sure I would take the proper branches and would stay on track.

Kurzfassung

Derzeitige statische Quellcodeanalysierungssoftware wird derzeit nicht ausreichend verwendet, weil sie von einer hohen Rate an falsch positiven und falsch negativen Fehlern betroffen ist. Das Aufkommen von großen Sprachmodellen (LLM) trainiert auf umfassenden Mengen an Quellcode, sogenannten „Models of Code“, bringt neue Hoffnung für die Erkennung von Fehlern und Sicherheitslücken. Das Erkennen von Fehlern bevor Quellcode kompiliert wird, könnte die Zeit, um Fehler zu finden und zu beheben drastisch reduzieren. LLMs sind prinzipbedingt nicht in der Lage ihre Antworten zu erklären, was sich negativ auf ihre vernommene Vertrauenswürdigkeit und Sinnhaftigkeit auswirkt. Eine Möglichkeit diese Nachteile zu umgehen sind Counterfactuals. Ein Counterfactual versucht eine Antwort eines Black Box Models, wie eines LLMs, zu erklären, indem es die Eingabe so lange verändert, bis das Modell zu einem anderen Ergebnis kommt. Diese Änderung wird als Grund für das ursprüngliche Ergebnis interpretiert. Ist ein Counterfactual zu weit von der ursprünglichen Eingabe entfernt, liefert es keine Informationen, da bei einer anderen Eingabe ohnehin eine andere Ausgabe erwartet werden kann.

Nach Counterfactuals zu suchen kann zeitaufwendig sein, da die fortlaufende Änderung der Eingabe einen exponentiellen Suchraum aufspannt. Diese Arbeit untersucht, wie unterschiedliche Suchalgorithmen und Konfigurationen den Suchraum beeinflussen, und verwendet als Vergleich eine exponentielle Suche. Die untersuchten Suchalgorithmen sind eine Gentische Suche (GS), eine Greedy-Suche, und eine Layer Integrierte Gradienten (LIG) Suche. Diese werden mit Maskierten Sprachmodellen und unterschiedlichen Models of Code, sowie mit verschiedenen Permutationsfunktionen und Tokenizern kombiniert.

Die Suche nach Counterfactuals leidet unter der suboptimalen Genauigkeit der derzeit zur Verfügung stehenden Models of Code, welche auf dem binären Klassifikationsproblem der Erkennung von fehlerhaften oder vulnerablen C++ Quellcode eine Genauigkeit von maximal 68.78% aufweisen. Trotzdem sind sowohl die GS als auch die Greedy-Suche schneller auf einer Zeit pro Counterfactual Basis als eine k-exponentielle, umfassende Suche (kEES). Die LIG-Suche ist 14-mal langsamer pro Counterfactual als kEES, da 90% der Suchläufe kein Counterfactual finden können. Wird die Zeit bis zum ersten gefundenen Counterfactual betrachtet, ist der schnellste Algorithmus die Greedy-Suche, gefolgt von der LIG-Suche, und der GS. kEES ist der langsamste Suchalgorithmus. Die Wahl des Models of Code, der Perturbationsfunktion und des Tokenizers hat großen Einfluss auf die Suchdauer pro Counterfactual.

Abstract

Current state-of-the-art static code analysis software is underused as it is plagued by a high rate of both false negatives and false positives. The emergence of Large Language Models (LLM) trained on comprehensive amounts of source code, so called models of code, provides new hope with detecting bugs and insecurities even before code is compiled, which could greatly reduce the time needed and the cost of finding and fixing bugs. However, LLMs are inherently unable to provide explanations for their outputs, which limits the trust that developers place in them, as well as their usefulness. One way to mitigate this drawback are explanations through counterfactuals. A counterfactual attempts to explain the reason for a certain decision of a LLM by perturbing the input such that the LLM comes to a different result. Then, this change might be the reason why the LLM arrived at its conclusion in the first place. Not all counterfactuals are equally useful. A counterfactual too far off the original input does not provide any information, as a completely different input may be expected to lead to a different output.

Searching for counterfactuals can be a time consuming task, as just applying changes to the input until a counterfactual is found spans up an exponential search space. This thesis investigates how different search algorithms and configurations affect this search space as compared to an exponential exhaustive search as benchmark. The search algorithms in this thesis are the genetic search, greedy search, and the layer integrated gradient (LIG) search. These are combined with masked language models and different models of code, Tokenizers and perturbation functions.

The search for counterfactuals suffers from the accuracy of current models of code, which, on a binary classification task of classifying source code as vulnerable or invulnerable, reach an accuracy of 68.78% or less on C++ source code. Even so, both the greedy search and a genetic search outperform the baseline k-Exponential Exhaustive Search (kEES) significantly in a time-per-counterfactual manner. The LIG search algorithm is 14 times slower than kEES, since 90% of its search run end without finding a counterfactual. When it comes to finding the first counterfactual, the greedy search algorithm is the fastest one, followed by the gradient informed search, the genetic search, and lastly the kEES. The choice of model of code, perturbation function, and Tokenizer greatly influences the search duration per counterfactual.

Contents

CHAPTER 1

Introduction

For certain types of bugs, current Machine Learning (ML) models of code can find potential bugs fast and efficiently, with processing times of less than 20 ms per file, and an accuracy of about 90% [35]. However, the output of ML models are inherently inexplicable for humans, which can reduce both the trust and usefulness of the output [15]. This can be a problem in software engineering, as knowing that a certain code is inefficient or insecure means little if the developer does not get any information about the reason or what specific line of code is to blame. Therefore, the developer might fail to take appropriate actions to resolve the underlying issue.

To make decisions of black box models, such as ML systems, explainable, the concepts of counterfactuals was introduced. For a given input i that yields an output o from a certain decision model, a counterfactual is a perturbation δ applied to i resulting in $\delta(i) = i'$, such that the same decision model comes to a different decision $o' \neq o$ when evaluating i'. Then, δ is considered to be the reason for the change in the decision made by the model.

As an illustrative example, one might imagine an individual who applies for a loan at a bank. The bank has to decide whether the credit rating of the individual is good enough. For this task, it employs some black box Artificial Intelligence (AI) system. This system is fed with the credit history and income of the individual. Imagine further that the credit history of the individual is considered bad and the income is 50.000€ per year. Now, the AI might determine that the individual should not be granted the loan. The individual might not be content with this answer and demand an explanation. With plain AI systems, such an answer might be impossible to give. However, one could search for counterfactuals, i.e. change the input until the output changes. One counterfactual might be an income of 50.000€ per year, but with a good credit history. In this case, the AI determines the individual to be credit worthy. Another might be a bad credit history, but an income of 70.000€. Those two counterfactuals are not equally useful, as it might be possible to increase ones income, but it is not as easy to change the past to attain a

better credit history. A possible search space spun up during this search might look as displayed in Figure 1.1.

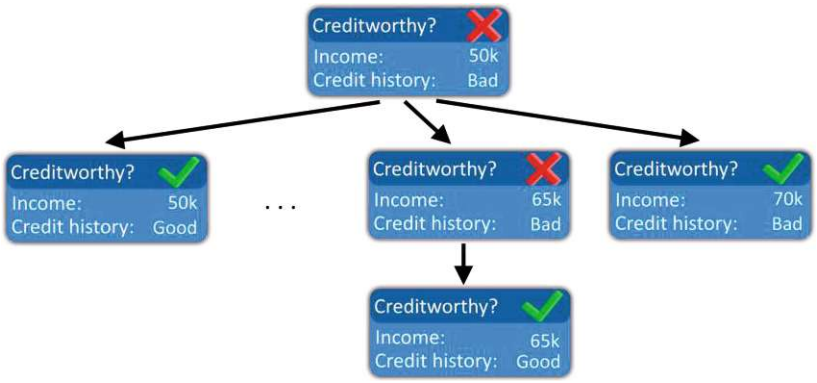

Figure 1.1: One possible search space in the search for a counterfactual to the creditworthiness example. Counterfactuals are marked with a green check mark. The leftmost two counterfactuals are of low usefulness, as they would require a change in the credit history. The center counterfactual needs an additional increase in income, and is therefore even less actionable. The counterfactual to the right requires an increased income, and is therefore more actionable in this scenario.

As explained in the example, not every perturbation δ is equally useful. If the resulting i' is too far off the original i, no actual information can be gained, as it is expected that a completely different input may result in a different output. Therefore, it is required from counterfactuals to be as close to the original input as possible, while still yielding different outcomes [17][34] The definition of closeness depends on the context. Furthermore, counterfactuals are expected to be actionable, i.e. the perturbation has to be a realistic change, that is still possible to achieve within the context of the input space [34]. To this end, [8] has proposed a technique to find counterfactuals for source code that appear "natural" to a human and are plausible, actionable, and consistent with respect to, and lie in the proximity of, the original input. This technique creates counterfactuals that were found useful by 83.3% of interviewed users. However, finding an appropriate perturbation using this method takes between 30 seconds and 10 minutes, depending on the size of the perturbation. These processing times do not allow for real-time applications and stem from the exponential complexity of the search for counterfactuals with respect to the input size. Therefore, it is crucial to find ways to shrink the search space and find efficient ways to search what remains of it.

This thesis aims at exploring ways to efficiently search through the most relevant parts of the search space to speed up the search for counterfactuals. To this end, a program is written that incorporates different search and perturbation strategies and which can apply those to different models of code. The resulting program is easily extendable and model agnostic as far as possible. Implemented search algorithms include a k-Exponential Exhaustive Search (kEES) as baseline, a Genetic Algorithm (GA), a Greedy Search

algorithm, and a Layer Integrated Gradients (LIG) search. Perturbation algorithms include token removal, mask replacement through the use of a Masked Language Model (MLM), and randomly selecting tokens from the input dictionary. The dictionary is created from the input through either a Line Tokenizer or a C++ Tokenizer using Clang. Resulting counterfactuals are evaluated for their similarity to the orignial input using the Jaro similarity [4]. Algorithms producing these counterfactuals are evaluated for the amount of counterfactuals and the time needed to produce them. The program also includes an array of different helper classes to be used in conjunction with the search algorithms, such as Tokenizers, Perturbers and Unmaskers.

Using this system, the following research questions shall be answered:

- **RQ1**: What is the search space for possible perturbations with respect to different search heuristics?

  The search space spanned up by employed search algorithms and perturbation functions might greatly differ according to the exact configuration used. While an exponential search has to iterate over all possibilities, heuristic search algorithms can focus their search on promising parts of the search space, thereby saving time and allowing for a more thorough exploration of these areas.

- **RQ2**: How can gradient information from the model of code influence the search space?

  Using gradient information could improve the performance of the search for counterfactuals, but the additional computing required could slow down the search such that heuristic search algorithms could be faster.

- **RQ3**: How do different perturbations functions influence the search space?

  Different perturbation functions paired with different Tokenizers might lead to a faster convergence on counterfactuals. Tokenizers yielding less but larger tokens might speed up the search, but the resulting perturbations might be too coarse to deliver counterfactuals.

CHAPTER 2

Background

This chapter explains the technical background of this thesis to the reader.

2.1 Transformer Architecture

The transformer architecture was introduced in the 2017 paper "Attention Is All You Need" [45]. In this paper, the authors propose a successor to the then state-of-the-art, the Recurrent Neural Network (RNN). RNNs have two major disadvantages, that limit their usage for large inputs and outputs. The first disadvantage is their sequential nature. A RNN requires a computation or time step for each token the input is split into. As each step requires a hidden state computed by the preceding state, this computation cannot be parallelized. This makes for a poor performance on large input sizes both during training and inference. The second disadvantage is the problem of vanishing or exploding gradients. Here, the efficacy of the backpropagation algorithm used to train the model declines when the model dimensions increase. This is due to the chaining of the derivatives of the loss functions of many different neurons results in many multiplications. If the factors tend to be lower than 1, the result will approach 0 (vanishing gradient), if they tend to be larger than 1, the resulting number will grow rapidly (exploding gradient). In either case, this means that the apparent contribution of a neuron to the output is either massively over- or underestimated [33].

Transformers claim to overcome both of these problems, allowing for fast learning on comparatively little training data [45], where the training on any one sample takes only a single computational step independent of the size of the sample. Inference, however, still requires a subsequent operation for each token. This is possible because the transformer can take any input of d_{model} tokens at one time. If the input is shorter than d_{model}, it needs to be padded with predefined padding tokens.

The original transformer as proposed in [45] consists of an encoder and a decoder, as displayed in Fig. 2.1. During training, the encoder is given the input sequence with prepended Start Of Sequence (<SOS>) token and appended End Of Sequence (<EOS>) token. The decoder is given the correct output sequence, which is prepended with a <SOS> token. The transformer is trained to output the output sequence without the <SOS> token, but with an appended <EOS> token. After one time step, both input and output sequence are processed by the transformer, finishing this training sample. Both the <SOS> and <EOS> tokens are special predefined characters in the dictionary of the transformer.

During inference, the encoder is given the input sequence with prepended <SOS> token and appended <EOS> token, as was the case during training. It calculates its output, which is passed on to the decoder. Furthermore, the decoder receives the <SOS> token. The decoder then generates the first token for the answer of the transformer from the <SOS> token and the output of the encoder. Then, in the second time step, the decoder is given the <SOS> token with the appended first token from the previous time step. The encoder does not need to recompute a new output, as the input does not change during inference for one sample. After computing the second output token, it is added to the input for the decoder, and the whole process is repeated until the transformer outputs the <EOS> token [45].

2.2 Generative Pretrained Transformer (GPT)

Through the use of unsupervised training, the need for large amounts of labeled data can be reduced. This was achieved in [36], where a transformer decoder variant was trained on a large corpus of continuous text data. The text data originated from the BookCorpus dataset, which contains over 7000 unpublished books [53]. The authors of [36] considered it important that the training data does not consist of random sentences or words, as they wanted to train their model on coherent text spanning over several sentences or paragraphs. This way, the model could be trained to generate continuous and natural text. After the unsupervised training, the model was fine-tuned on tasks such as natural language inference, text classification, question answering, and others, using datasets specific for each task. The resulting type of model is today known as Generative Pretrained Transformer (GPT).

As of the writing of this thesis, the described approach, albeit heavily modified and expanded, is the state-of-the-art for Large Language Model (LLM), such as GPT-4 [1], BERT [11], and LLaMA [44], among others. However, due to the large size of the referenced models, the BookCorpus dataset plus the fine tuning data sets would not be enough. Therefore different and much larger training data sets are used. The exact data used and ways in which it is used are often kept secret by the organizations creating these models [3][1]. It is known, however, that next to the quantity, the quality of the training data is important. This means that textual training data should be written by humans and should have undergone an editing process. Wikipedia, peer-reviewed journal

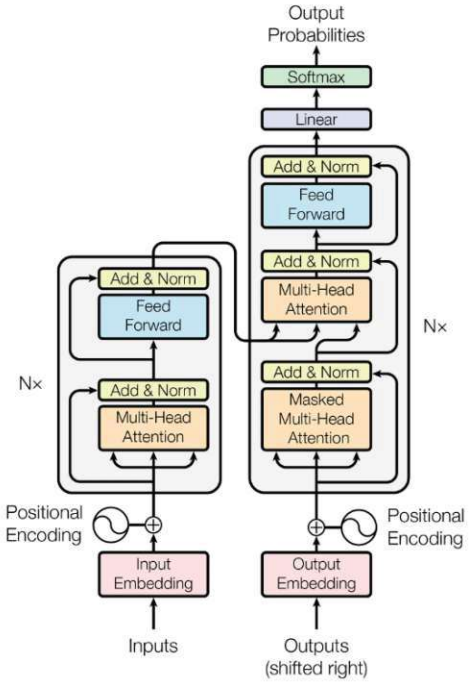

Figure 2.1: The transformer architecture as presented in the paper "Attention Is All You Need". The left side is the encoder which takes the input with start and end tokens. The right side is the decoder. During training, it is fed the expected output, during inference it is fed a start token in the first time step and its own output afterwards. Image from [45].

articles, and books satisfy these requirements, and are therefore considered high quality training data [3]. More recent models such as GPT-3, GPT-4, and others, also undergo an additional training step called Reinforcement Learning from Human Feedback (RLHF). Here, humans validate the responses to input prompts to certain questions aimed at producing unwanted answers, such as how to build a bomb or how to synthesise dangerous chemicals. This step is especially aimed at preventing the model from generating offensive outputs and to reduce bias introduced by the training data [1]. However, RLHF can overshoot its mark and lead to degraded performance and incorrect responses [38].

Model sizes have increased drastically over the different generations of models. The original GPT [36] model has a size of 110M parameters, GPT-2 uses 1.5 billion parameters [50], and GPT-3 175 billion parameters [14]. OpenAI does not disclose the dimensions of GPT-4 [1]. However, [6] found ways to extract hidden information from transformer models using only conventional API access, but choose to employ this attack only against models when given permission from their owners. Therefore they did not find or publish any previously undisclosed information. Modern GPT models are able to emulate a conversation in natural language with humans in different languages, and perform well in a variety of exams aimed at humans. GPT-4 is claimed to perform amongst the top

10% of test takers of the bar exam[1], but also sometimes "hallucinates" and produces unreliable or wrong outputs [1]. Other models such as CodeBERT [12] are also trained specifically on programming language tasks. The extensive training of GPTs and their resulting world knowledge models allows them to be fine tuned to previously unknown tasks on new data with relatively little additional training [26].

2.3 Code Review Models

Automated code review and bug finding has received increased attention lately, especially with the increased popularity and availability of deep learning tools. Such tools aim to overcome both the high false-positive and false-negative rates of static code analysis tools, which is currently a factor why they are underused [23][41]. Current state of the art ML models of code are able to find bugs fast and accurately, e.g. DeepBugs [35], which claims an accuracy of between 84.23% and 94.53% on name based bugs in JavaScript code, and a processing time of less than 20ms per file. However, these systems cannot tell how they have reached the conclusion that certain parts of source code contain bugs and are often limited in the type of bugs that can be detected. The inability of a model to explain its results limits its usefulness and the trust that developers place in it [15].

2.4 Counterfactual Search Heuristics

To find counterfactuals, this thesis explores different search strategies.

2.4.1 Genetic Search (GS)

A GA or Genetic Search (GS) is an approach to solve complex problems inspired by the process of natural genetics and natural selection in the real world. These types of algorithms require little problem information [10]. Some form of loss or fitness function, however, is required. This means that e.g. a plain binary problem is not solvable by this type of algorithm.

During the runtime of a GA, several different attempted solutions compete for survival. Each of these attempts are represented by a binary string of the same arbitrary length and structure that represents the parameters or variables of the problem. As an example, one might imagine a GA trying to minimize the surface area of a cylinder, while keeping the volume above a value of 1000. The fitness function might be the inverse of the surface area of the cylinder, such that a high surface area corresponds to a low fitness. If the volume is lower than the specified minimal value, the fitness might be zero. A solution candidate might consist of an eight bit binary string, where the first four bits encode the height and the last four bits encode the radius of the cylinder.

1101 0110

[1] https://www.americanbar.org/groups/legal_education/resources/bar-admissions/bar-exams/

$$height = 13 \quad radius = 6$$

The example candidate has a height of 13 and a radius of 6, resulting in a volume V of ≈ 1470.27, and a surface area A of ≈ 716.28. Its fitness is $1/A$, which is ≈ 0.001396.

During the runtime of the GS, the candidates in the gene pool are evaluated for their evolutionary fitness by evaluating the loss function given the parameters of the candidate. Depending on this fitness, candidates are either given the opportunity to reproduce or are removed from the set of possible solutions, i.e. killed.

Genetic algorithms are structured in phases:

- **Phase 1: Initial Population**
  At the beginning, a gene pool of candidates with random parameters, i.e. binary strings, is created.
- **Phase 2: Evaluation**
  This phase marks the begging of an iteration or generation. All candidates in the gene pool are tested against the problem and their fitness is calculated.
- **Phase 3: Selection**
  Candidates are selected for survival. Candidates with a high fitness are more likely to survive, but still have a chance of being removed from the gene pool. This phase is crucial to limit the diversity in the gene pool to promising candidates.
- **Phase 4: Reproduction**
  The free space generated by the killing of unfit candidates is now filled up by letting candidates reproduce. The higher the fitness of a candidate, the higher is its chance for reproduction. The resulting children hopefully inherit promising traits from their parent or parents.
- **Phase 5: Mutation**
  To introduce and increase diversity in the gene pool, randomly selected candidates are mutated. This phase concludes an iteration, and the algorithm starts over at Phase 2 until the iteration limit is reached or a candidate is found that performs well enough.

Selection

In the selection phase of a GA, it is decided which solution candidates may live on to the next generation. This is done by comparing their fitness with the competition. Higher fitness means greater chance for survival. However, to allow for greater diversity and thereby lower chance of getting stuck in a local optimum, even the fittest of candidates might be killed, and the candidates with the lowest fitness might survive. The actual selection of candidates to be killed is usually done through tournament selection or roulette wheel selection. In some implementations, the candidate with the highest overall

fitness is spared from the selection and will always survive. The ratio of candidates killed from the gene pool per generation is usually predefined before the start of the GA.

Reproduction

Reproduction is done through crossover when using two parents or by copying and mutating a candidate when a single parent is used.

In crossover, genes from two parents are merged to produce one or two new solution candidates. The genes are chosen through some random selection from either parents binary string. The parents are usually chosen through tournament selection or roulette wheel selection with respect to their fitness. Applied to the cylinder example above, crossover of parent candidates C_a and C_b producing two children C_{ab} and C_{ba} might look as follows:

$$C_a : 110|1 \quad 0110 \quad \Longrightarrow \quad C_{ab} : 110|0 \quad 1100$$
$$C_b : 100|0 \quad 1100 \quad \Longrightarrow \quad C_{ba} : 100|1 \quad 0110$$

Where the vertical line | represents the pivot point. Bits before this point are taken from one parent, bits after this point from the other parent. The position of the pivot point is random and different for each crossover operation. There may also be a second pivot point, after which the parent who's bits are taken from is swapped again.

Mutation

Both the reproduction step and the mutation step aim at increasing the diversity in the gene pool. Mutating a candidate means to change at least one of its parameters at random. Applied to the cylinder example above, mutation of candidate C_a might look as follows:

$$C_a : 1101 \quad 0\underline{1}10 \quad \Longrightarrow \quad C_{a'} : 1101 \quad 0\underline{0}10$$

Where the underlined bit is flipped. A mutation could also affect more than one bit and is not limited to bit flipping.

Both mutation and crossover can produce new candidates that perform badly. This does not impact the overall performance of the GA, as these candidates will likely be killed in the next generation. However, since new candidates are created from already well performing candidates, it is more likely that their children will perform as good as, if not better than, the parents. Over time, this leads to the gene pool being occupied largely by good candidates, and the fitness of the total gene pool should increase [10].

GAs are not always restricted to the use of variables and parameters encoded in binary strings. Different approaches, such as graph encoding as used by NeuroEvolution of

Augmenting Topologies (NEAT) [42], exist. In NEAT, the topology of a Neural Network (NN) is modified randomly. Initially, a NN in NEAT starts out just with as many input and output neurons as the problem to solve requires. Mutations include the addition or removal of hidden neurons and connections between neurons, as well as the modifications of weights associated to the connections between neurons. During crossovers, the graphs of two parent NN are merged to produce one child. This allows for complex NNs, which can even have memory by transitively connecting the output of a neuron to its input. Still, the principles of GAs are followed, as NEAT starts out with an initial population, and aims to improve this population through selection, crossover and mutation [42].

2.4.2 Greedy Algorithm

A greedy algorithm takes at each decision point the option which promises the highest local improvement. This strategy does not always lead to an optimal solution, but might lead to a solution that is good enough and faster than e.g. an exponential search [5]. It requires some heuristic, which can give an estimate of the magnitude of improvement for any given choice at any decision point. By following many local optima, it is hoped that eventually a global optimum will be reached. However, in general such a guarantee cannot be given [9].

2.4.3 k-Exponential Exhaustive Search (kEES)

A kEES is an exhaustive search algorithm, that searches the whole search space of a given problem. Since for any nontrivial problem this search space is too large to completely iterate through in a reasonable time, kEES aborts the search after reaching a search depth of k. This allows the algorithm to evaluate all possible solutions within this depth, but prohibits it from focusing its search at promising avenues. In contrast to the GS and greedy algotrithms, an advantage of the kEES is that it does not require a loss or fitness function. Furthermore, it does not require any heuristics, structures in, or understanding of the problem and the effect of individual parameters on the problem, as all possibilities will be evaluated anyway [32]. Therefore kEES can in theory be used on any problem, but its runtime might be prohibitively slow for problems with nontrivial branching factors.

2.4.4 Layer Integrated Gradients

This thesis uses the LIG implementation from the Captum Library[2] to make the results of the models of code interpretable [25]. Captum is open source[3] and supports an array of different technologies intended to solve this task. It also provides tools to visualize its results.

LIG is based on Integrated Gradients (IG), but assigns an importance score to the inputs or outputs of each layer in the NN [25]. IG works by integrating the gradients of the

[2]https://captum.ai/api/layer.html#layer-integrated-gradients
[3]https://github.com/pytorch/captum

output of a NN with respect to the input. It does this by computing the average gradient while the input is shifted from the original input to some baseline. This baseline provided by the user is usually set to zero or an equivalent value. There are no modifications required to the NN for IG to work [7].

2.5 Evaluating Counterfactuals

Counterfactuals are evaluated for their closeness to their original input using the Jaro Distance [22]. This distance is a dimensionless number. A value of 1 means equals strings, and 0 means no similarity between two given strings. The Jaro similarity function counts the number of matching characters c and the number of pairs of adjacent characters t which are swapped in both input strings $s1$ and $s2$. Then, the distance is calculated as follows:

$$Jaro(s1, s2) = \frac{1}{3}(\frac{c}{|s1|} + \frac{c}{|s2|} + \frac{c-t}{c})$$

where $s1$ and $s2$ are the strings to be compared, and $|s|$ gives the number of characters in string s [30].

For the context of this thesis, the actual value of a string similarity of a counterfactual to its respective input is only relevant when it is compared to the similarities of other counterfactuals. This thesis assumes that if two counterfactuals have a different Jaro distance, the one with the higher distance is closer to the input and therefore more useful to the user.

The jellyfish Python library[4] is used to compute the Jaro distance in this thesis.

[4] https://jamesturk.github.io/jellyfish/

CHAPTER 3

Study Design

This chapter shows how this thesis is designed to answer the research questions.

3.1 Overview

To investigate how different settings influence the counterfactual search space, this thesis uses an array of interchangeable components in a way to produce comparable results. Each of these components can influence the search space to make the search faster, more thorough, more efficient, or to change it in some other way. A broad overview of the types of components is presented in Figure 3.1 and the following text. A more detailed explanation is given in Section 3.2.

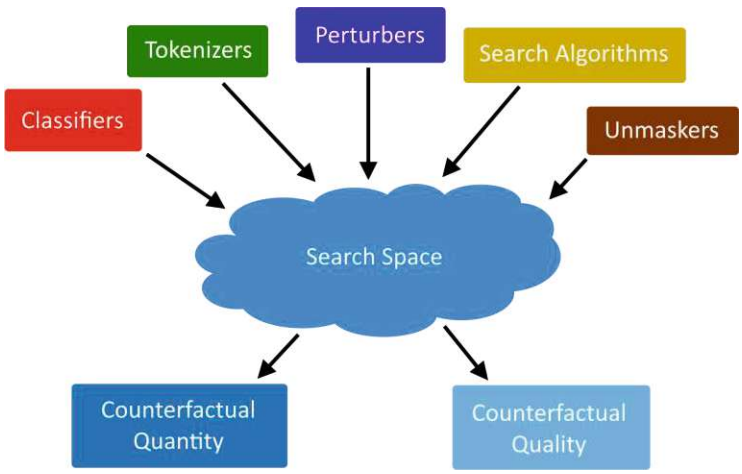

Figure 3.1: The exact configuration of the search methods used influences the search space, which in turn influences the number and types of produced counterfactuals.

The search space is spun up by the following components:

- **Classifiers:** In the context of this thesis, a classifier takes some source code snippet and classifies it into one of two categories, either vulnerable or not vulnerable. Classifiers can also provide metadata to decisions, such as a confidence score. The accuracy of classifications and of the provided confidence score can help the search with finding better counterfactuals, and to do so faster.

- **Tokenizers**: A Tokenizer splits the source code snippet into a list of tokens, and can convert a list of tokens back into a source code snippet. Such a list of tokens can be changed by a Perturber. Tokenizers can split the input by lines, words, or some other means, and can therefore decide on the granularity of a change and have great influence on the width of the search space.

- **Perturbers**: Perturbers change the list of tokens from some source code snippet by removing, adding new or existing tokens, or changing entries in this list. The resulting changed source code snippet can then lead to a different classification and therefore a counterfactual.

- **Search Algorithms**: The employed search algorithm decides which avenues within the search space should be explored. This search can be guided through metadata provided by the classifier or it can perform an exhaustive exploration. The efficiency of a search algorithm, and its capability to use metadata from the classifier, has a strong effect on the speed of the search and the number of produced counterfactuals.

- **Unmaskers**: If the employed Perturber allows it, it can add predefined mask tokens to the list of tokens. These mask tokens will then be unmasked, i.e. replaced with source code appropriate for the location of the mask token, by an Unmasker. If an Unmasker is able to fix vulnerabilities in a source code snippet, this can easily lead to the discovery of a counterfactual, or at least guide the search into a promising direction.

Together, these components can change how many counterfactuals can be found, and how fast the search is. They also affect the quality of the counterfactuals. The aim of this thesis is to find how different implementations of these components change the search space differently, and which perform better than others in some regard.

3.2 Approach

This section describes the approach taken in developing the counterfactual generator.

3.2.1 Project Structure

As is common in the field of ML, the software behind this thesis is implemented in Python and makes use of an array of different libraries, such as the Transformers library[1], NumPy[2], and others. In order to make the system searching for counterfactuals as easily expandable as possible, it is structured in a modular way. This means, that each component is abstracted into its own abstract superclass, from which individual implementations are derived. Components include Classifiers, Tokenizers, Perturbers, Search Algorithms, and Unmaskers.

Classifiers

In the context of this section, Classifiers are wrapper classes around models of code. These are given some source code and are asked to provide a classification. If the underlying model is able to supply one, a confidence score is provided, too. The abstract superclass is the `AbstractClassifier` class, as given in Listing 3.1.

```
1 class AbstractClassifier:
2     def classify(self, source_code: str) -> (bool, float):
3         """Evaluates the input and returns a tuple with (result,
              confidence)"""
4         raise NotImplementedError
```

Listing 3.1: The abstract superclass for all Classifiers.

This class contains a single function `classify(self, source_code: str)`, which has to be implemented by its inheritors. It returns a tuple containing the classification and the confidence as float. The classification is returned as a boolean value. Typically, `False` means secure code and `True` means unsecure code. The actual value of this classification has little consequence, as only a change of this value over different inputs contains any information with respect to the search for counterfactuals. However, the actual classification is checked in the `AbstractSearchAlgorithm`, and searches on code that is classified as secure are aborted.

One example implementation is the `PLBartClassifier`, which uses the PLBart [2] model as described in Section 4.1.2 to classify input strings. Its source code is displayed in Listing 3.2.

```
1 class PLBartClassifier(AbstractClassifier):
2     path = "uclanlp/plbart-c-cpp-defect-detection"
3     tokenizer = AutoTokenizer.from_pretrained(path)
4
5     def __init__(self, device):
```

[1] https://pypi.org/project/transformers/
[2] https://numpy.org/

```
6         self.model =
              PLBartForSequenceClassification.from_pretrained(self.path,
              output_attentions=True).to(device)
7         self.device = device
8
9     def classify(self, source_code: str) -> (bool, float):
10        """Evaluates the input and returns a tuple with (result,
              confidence). A result of 0 means secure code,
11        1 is insecure"""
12        inputs = self.tokenizer(source_code, return_tensors="pt")
13        input_ids = inputs["input_ids"].to(self.device)
14        attention_mask = inputs["attention_mask"].to(self.device)
15
16        with torch.no_grad():
17            logits = self.model(input_ids=input_ids,
                  attention_mask=attention_mask).logits
18            clazz = logits.argmax().item()
19            return int(clazz) == 0, float(logits[0][clazz])
```

Listing 3.2: The implementation of the `PLBartClassifier`. Note that the input tensors are moved to `device`, which is either a Graphics Processing Unit (GPU) or Central Processing Unit (CPU), depending on the software and hardware available.

Additionally, the `AbstractClassifier` class contains several abstract methods to provide access to raw data and the models tokenizer to be used by the Captum library to perform the LIG search. These methods are omitted for brevity, but include getters for the IDs of begin-of-string, padding, and end-of-string tokens, as well as methods to map tokens to their ID and vice versa and to obtain the embeddings of the model. An overview over all Classifiers used in this thesis is given in Figure 3.2.

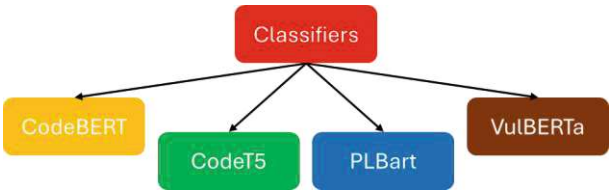

Figure 3.2: An overview over all Classifiers used in this thesis.

Tokenizers

The Tokenizers split the input string into several tokens, and in the process of this splitting create a dictionary, which is a list containing at least all tokens in the input. The ID of a token is its index in the dictionary list. Note that these Tokenizers may differ from Tokenizers used by the actual models of code as given in Listing 3.2, line 10. The granularity of the tokens depends on the actual implementation. Each token is assigned an unique token id. Furthermore, a Tokenizer can provide the string

represented by a list of token IDs. The abstract superclass for all Tokenizers, the `AbstractTokenizer` class, is presented in Listing 3.3. This class contains an abstract function `tokenize(self, source_code: str)`, which has to be implemented by its inheritors. This function splits the input string into tokens and returns a tuple containing the number of tokens in the input and the generated dictionary. Furthermore, the `AbstractTokenizer` contains the functions `to_string(self, dictionary: List[str], tokens: List[int]) -> str` to reconstruct a string from a list of token IDs, and `to_string_unmasked(self, dictionary: List[str], tokens: List[int], replace_with_mask: int) -> str`. The latter function not only reconstructs the string from the list or token IDs, but, if it finds the predefined `AbstractUnmasker.MASK_INDEX` as id, or if the index in the `tokens` list matches the `replace_with_mask` parameter, replaces that token with the result of the `AbstractUnmasker` given in the `__init__(...)` function.

```
1  class AbstractTokenizer:
2      EMPTY_TOKEN_INDEX = -2
3
4      def __init__(self, language: Language, unmasker: AbstractUnmasker
           | None = None):
5          self.language = language
6          self.unmasker = unmasker
7          if unmasker is None:
8              self.mask = None
9          else:
10             self.mask = unmasker.get_mask()
11
12     def tokenize(self, source_code: str) -> (int, List[str]):
13         """Returns a tuple containing the number of tokens in the
               document, and a list of all available words,
14         including words not in the document"""
15         raise NotImplementedError
16
17     def to_string(self, dictionary: List[str], tokens: List[int]) ->
           str:
18         perturbed_sequence = []
19
20         for i in tokens:
21             if i == AbstractUnmasker.MASK_INDEX:
22                 perturbed_sequence.append(self.mask)
23             elif i == self.EMPTY_TOKEN_INDEX:
24                 pass
25             else:
26                 perturbed_sequence.append(dictionary[i])
27
28         return format_code(' '.join(perturbed_sequence), self.language)
29
30     def to_string_unmasked(self, dictionary: List[str], tokens:
```

```
        List[int], replace_with_mask: int = -1) -> str:
31      perturbed_sequence = []
32
33      for i in tokens:
34          if i == self.EMPTY_TOKEN_INDEX:
35              pass
36          elif i == replace_with_mask or i ==
                AbstractUnmasker.MASK_INDEX:
37              code = format_code(self.to_string(dictionary, tokens),
                    self.language, self.mask)
38              replacement = self.unmasker.get_mask_replacement(code)
39              perturbed_sequence.append(replacement)
40          else:
41              perturbed_sequence.append(dictionary[i])
42
43      return format_code(' '.join(perturbed_sequence), self.language)
```

Listing 3.3: The abstract superclass for all Tokenizers, the `AbstractTokenizer`.

One example implementation of the `AbstractTokenizer` is the `LineTokenizer`. This Tokenizer splits the input at every newline `'\n'` character and therefore produces one token per unique line in the input. An overview over both Tokenizers used in this thesis is given in Figure 3.3.

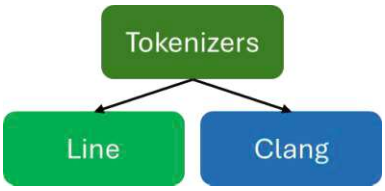

Figure 3.3: An overview over both Tokenizers used in this thesis.

Perturbers

The task of a Perturber is to change a given source code, such that a Classifier might classify it differently than the original input, thereby generating a counterfactual. To help with this task, counterfactual candidates are stored in the program as instances of the `Entry` class or variations of it. Each such instance has a `classification`, `fitness`, and `document_indices` variable. The `classification` and `fitness` variables store meta data according to the search algorithm, or might be left unused. The `document_indices` variable stores a list of the token IDs in the correct order making up the source code of this candidate. By removing or adding an index to this list, a token can be removed or added to a candidate. It is the task of a Tokenizer to provide the indices for the list and to convert this list back into a string. To keep track of the search history of a counterfactual, each `Entry` object also contains the `number_of_changes`variable which counts the number of changes to the original input,

and the `changed_values` set, which contains the token IDs of all tokens that were changed from the original input.

The base class for all Perturbers is the `AbstractPerturber` class, as displayed in Listing 3.4. It contains a function `perturb(self, source: List[int], dictionary_length: int) -> List[int]`, which takes a list of token IDs and returns a new and perturbed list from the `source` parameter, with perturbances according to the implementation of `perturb_in_place(...)`. Two further functions are abstract. The first abstract function, `perturb_in_place(self, source: List[int], dictionary_length: int)`, applies the perturbation to the `source` parameter in-place, without creating a new list. The second abstract function, `perturb_at_index(self, index: int, source: List[int], dictionary_length: int)`, perturbs the token ID at `index` in the `source` list.

```
1  class AbstractPerturber:
2      def perturb(self, source: List[int], dictionary_length: int) ->
           List[int]:
3          result = [*source]
4          self.perturb_in_place(result, dictionary_length)
5          return result
6
7      def perturb_in_place(self, source: List[int], dictionary_length:
           int):
8          raise NotImplementedError
9
10     def perturb_at_index(self, index: int, source: List[int],
           dictionary_length: int):
11         raise NotImplementedError
```

Listing 3.4: The abstract superclass for all Perturbers, the `AbstractPerturber`.

An example implementation of the `AbstractPerturber` is the `RemoveTokenPerturber`, as displayed in Listing 3.5. This implementation removes a random token from the `source` list in the `perturb_in_place(...)` function, and replaces the token at `index` in the `source` list in the `perturb_at_index(...)` function with the special token `AbstractTokenizer.EMPTY_TOKEN_INDEX`, which is a placeholder for an empty string.

```
1  class RemoveTokenPerturber(AbstractPerturber):
2      def perturb_in_place(self, source: List[int], dictionary_length:
           int) -> int:
3          index = int(len(source) * random.random())
4          original = source[index]
5          del source[index]
6          return original
7
8      def perturb_at_index(self, index: int, source: List[int],
           dictionary_length: int):
```

```
9        source[index] = AbstractTokenizer.EMPTY_TOKEN_INDEX
```

Listing 3.5: The implementation of the `RemoveTokenPerturber`.

An overview over the Perturbers used in this thesis is given in Figure 3.4.

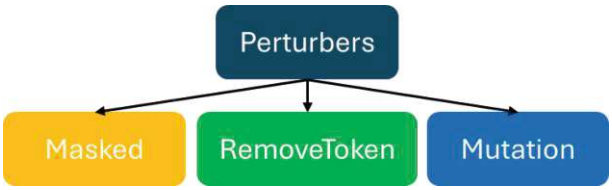

Figure 3.4: An overview over the Perturbers used in this thesis.

Search Algorithms

The abstract class `AbstractSearchAlgorithm` is a wrapper around the actual implementations of the different search algorithms used in this thesis. This is done to provide a uniform interface between the search algorithms and the rest of the framework. `AbstractSearchAlgorithm` contains an abstract function `perform_search(...) -> List[Counterfactual]`, which has to be implemented by its inheritors. This function does the actual searching for counterfactuals, and returns a list of all counterfactuals found during the search. It should not be called by the user. The `search(...) -> SearchResult` function is the public facing search function of this abstract class. It does some preprocessing steps common to all implementations of search algorithms, such as classifying the original input source code, and then calls `perform_search(...)`. The returned `SearchResult` object contains a possibly empty list of `Counterfactuals`, the time needed to complete the search, the names of all components assisting in the search, as well as the original input. If unrecoverable errors occur during the search, these errors are also stored in the `SearchResult` object, as discussed in Section 4.2.4. An overview over the search algorithms used in this thesis is given in Figure 3.5.

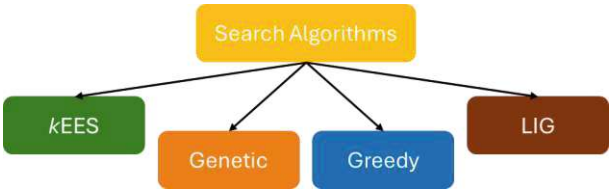

Figure 3.5: An overview over all search algorithms used in this thesis.

Unmaskers

Unmaskers turn predefined mask tokens into source code. This can be done through the use of a MLM. The abstract superclass for all Unmaskers is the `AbstractUnmasker`. This class consists of two abstract functions. `get_mask(self) -> str | None`

returns the mask token that the MLM uses. `get_mask_replacement(...) -> int` is given a source code as string which may contain a mask token. It queries the mask replacement from the MLM, and appends this new token to the dictionary. Lastly, it returns the index in the dictionary of this new token. If the given source code does not contain the mask token, this function will raise an exception, which the search algorithm has to handle, e.g. by pruning the affected search branch.

One example implementation is the `CodeBertUnmasker`. It uses the CodeBERT-MLM as described in Section 4.1.1 to replace `<mask>` tokens with source code. CodeBERT-MLM returns a list containing several possible mask replacements, which are sorted such that the replacement with the highest confidence is at index zero. This particular implementation always chooses this most promising replacement.

There is also a `NoOpUnmasker`, which is used when the `MaskedPerturber` is not used, and therefore no mask tokens are introduced. This implementation does nothing.

An overview over both Unmaskers used in this thesis is given in Figure 3.6.

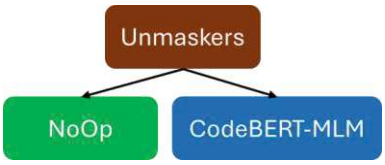

Figure 3.6: An overview over all Unmaskers used in this thesis.

3.2.2 Search Algorithms and Heuristics

This section describes the implementation of the following search algorithms.

Genetic Search

This is the implementation of the GS algorithm as described in Section 2.4.1. The Python class containing the source code is called `GeneticSearchAlgorithm` and inherits from `AbstractSearchAlgorithm` as presented in Section 3.2.1. It overwrites the `perform_search(...)` function, which initiates the GA. To instantiate the genetic search algorithm and use it to search for counterfactuals for some given C++ source code, the Python code in Listing 3.6 can be used.

```
1 unmasker = CodeBertUnmasker(device)
2 tokenizer = ClangTokenizer(unmasker)
3 classifier = PLBartClassifier(device)
4 perturber = MaskedPerturber()
5
6 search_algorithm = GeneticSearchAlgorithm(tokenizer, classifier,
      perturber, iterations=30, gene_pool_size=10)
7
8 results = search_algorithm.search(some_cpp_code)
```

Listing 3.6: The Python code to instantiate the genetic search algorithm with a gene pool size of 10 and an iteration limit of 30. Line eight starts the search for counterfactuals for some C++ code.

For this implementation, the fitness function is derived from the confidence of the model in a prediction. The GS expects a stable fitness function, i.e. if a perturbation increases the fitness of a candidate, then this perturbation must really have improved the candidate. However, the models tested in this thesis do not always satisfy this criteria as discussed in Section 6.3.

Initial Population The GA starts by creating a random initial population of a predefined size. The population consists of `Entry` objects, as described in Section 3.2.1. Each of these objects is initialized with the same tokens as the original input, and is then perturbed according to the Perturber passed in the `__init__(...)` function.

Selection The fitness of each candidate corresponds to the difference in confidence of its classification to the confidence of the classification of the original input, and the difference in the number of tokens of the candidate to the original input. This means, that if the original input was classified as `defective` with a confidence score of 0.78 and a candidate is classified as `defective` with a confidence score of 0.38, its fitness is $0.78 - 0.38 = 0.4$. The idea behind this fitness function is that if the model is less certain that some string is defective, the perturbations applied to that string have made the source code less defective, and the search heads in the right direction. The second term of the fitness function calculates the relative length difference of a candidate to the original input. This is computed by subtracting the quotient of the absolute token difference and the token length of the original input from one. E.g. if the original input consists of 10 tokens and the current candidate has 12 tokens, this would result in a penalty of $1 - (abs(10 - 12)/10) = 0.8$. To reduce the influence of this penalty term, it is then set to be the mean of one and the penalty term, e.g. a penalty of 0.8 is transformed into $(0.8 + 1)/2 = 0.9$. The penalty term should keep counterfactuals actionable and close to the original input by punishing possible solutions that stray off too far from the original input. The total fitness is the multiplication of the confidence score difference and the penalty, applied to the examples above, this would result in a fitness of $0.4 * 0.9 = 0.36$. Counterfactuals found during this evaluation step are removed from the gene pool to free up room for new candidates.

After the evaluation phase, the gene pool is culled. Candidates to be killed are chosen via a roulette selection, such that the fitness of a candidate corresponds to its chance for survival. It is ensured that the best performing candidate will always survive. The number of killed candidates is calculated from the `kill_ratio` parameter and the gene pool size.

Reproduction The space in the gene pool freed up by the selection phase is filled by either reproduction or mutation. One half of the freed up space is filled up through mutation using one parent, the other half is filled up through reproduction using two parents. In either case, the parents are selected through a roulette selection.

Mutations are done by copying the selected parent and perturbing the copy through the perturber. This operation leaves the parent unchanged.

Reproductions are done by choosing two parents with a roulette selection, and then computing a random pivot index from the token list of the parent with the lowest number of tokens. All tokens up to this index are taken from the parents with less tokens, all tokens afterwards are taken from the other parent. This operation produces one new candidate and leaves both parents unchanged.

After the reproduction phase, the iteration is complete and the algorithm starts over at the selection phase until the iteration limit is reached.

Greedy Search

This section discusses the implementation of a Greedy Search algorithm as described in Section 2.4.2. The Python class `GreedySearchAlgorithm`, which inherits from `AbstractSearchAlgorithm`, contains the source code for this implementation.

It works by firstly creating an initial population, and then successively iterating over a promising candidate, until a counterfactual is found or the iteration limit is reached. The initial population is created by instantiating a candidate for each token in the input. This corresponding token is then perturbed by the given Perturber. Then, each such candidate is classified by the given Classifier. Counterfactuals are removed from the population and stored in a `counterfactuals` list. For all other candidates, the heuristic value is calculated by subtracting their confidence score from the confidence score of the original input. All candidates are inserted into a list. For each search iteration, a candidate is selected from the list through a roulette selection, such that the best candidate has the highest chance for selection, but worse candidates also have a chance. This allows for a broader search space and also gives less promising candidates the possibility to prove themselves. The selected candidate is copied, and this copy is perturbed and evaluated. If a counterfactual has been found, the original best candidate is removed from the population, and the copy is added to the `counterfactuals` list. Otherwise, the original selected candidate remains in the list, and the copy is added to the list with its heuristic value calculated as described above.

k-Exponential Exhaustive Search

This is the implementation of the kEES algorithm as described in Section 2.4.3. The Python class containing the source code is called `KExpExhaustiveSearch` and inherits from `AbstractSearchAlgorithm` as presented in Section 3.2.1.

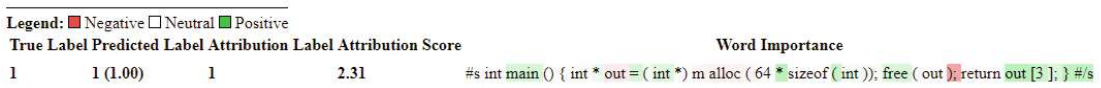

Figure 3.7: A visualization of a code sample evaluated by the Captum library. Tokens colored in green add to the result, which in this sample is 1 for unsecure code. Tokens colored in red subtract from the result.

At the start of the *k*EES, a new candidate is created for each input token. Then, the token associated with each candidate is perturbed according to the Perturber, and the candidate is classified by the Classifier. If the candidate is a counterfactual, it is removed from the set of candidates and added to the list of counterfactuals. Otherwise, new candidates are created from the current one for each of its tokens that were not modified by it or by its predecessors. This process is repeated until the iteration limit k is reached, at which point all counterfactuals are returned.

Layer Integrated Gradients

As discussed in Section 2.4.4, LIG, as implemented by the Captum library, is used to make the models of code interpretable. The baseline is constructed as a tensor with the same length as the input. The begin-of-string and end-of-string tokens, typically `<s>` and `<\s>`, respectively, are placed at the same indices as in the input. All other indices of the baseline tensor are filled with padding tokens, which is typically `<pad>`.

Captum provides tools to visualize the attributions of tokens to the final result. A screenshot of such a visualization is given in Figure 3.7. It shows the relative attribution of each token through a color coding. Tokens that add to the result are colored green, tokens that subtract from the result are colored red. The intensity of the color corresponds to the magnitude of contribution of the token.

Not all models can be used with Captum without further ado. The complete pipeline for the VulBERTa-MLP model does not output the raw logits tensor of the model, instead the results are mapped in a dictionary for each class, and these dictionaries are put in a list which is returned to the user. As Captum needs a raw logits tensor, the pipeline has to be bypassed and the model has to be used directly. The pipeline performs some preprocessing on the input before passing it to the model, so bypassing it might affect its accuracy. When Captum is used with the CodeT5 model, the program execution hangs inside the `forward(...)` call of the model and does not terminate. Therefore, the LIG approach is unusable with this model.

Counterfactuals are generated through the use of LIG by computing the attributions of each input token for the target output class. As counterfactuals have per definition a different classification than the input, the target class is the opposite of the input class. Then, the token with the highest attribution is deleted, and the resulting tokens are classified. If this results in a counterfactual, the search is terminated and this counterfactual is returned. Otherwise, the search continues. Attributions can be recalculated after each

iterations if desired. If not, the attributions for the original input are used in each search iteration. Note that here, the tokenizers are not those discussed in Section 3.2.1, but the tokenizers downloaded from Hugging Face associated with each model of code.

This approach means that a LIG search will yield at most one counterfactual.

CHAPTER 4

Evaluation

This chapter describes how the models used in this thesis were selected, and how the resulting system was evaluated.

4.1 Model Selection

Depending on the specific algorithm used to search for counterfactuals, different requirements must be met by the evaluation model. Some search algorithms, such as a GS algorithm, require the model to output a confidence score as a real number in addition to the classification. This is needed as the confidence can be translated to the evolutionary fitness of a gene as described in Section 2.4.1. If during the GS some gene attains the same classification than the original input but at a lower confidence, it can be assumed that it is closer to flipping the decision of the evaluation model, and should therefore be explored further. Other search algorithms, such as the kEES, do not rely on a confidence score, as they explore the whole search space exhaustively up to a depth of k in any case. This section describes the models used in this thesis, some of their properties, and the source code needed to get a classification for a given input string.

4.1.1 Microsoft CodeBERT and CodeReviewer

The CodeBERT model by Microsoft [12] is available on Hugging Face[1], and an array of testing and training scripts is available on GitHub[2]. There exist several derivates of CodeBERT. One derivate is CodeReviewer [26]. It supports quality estimation, comment generation, and code refinement tasks.

[1] https://huggingface.co/microsoft/codebert-base
[2] https://github.com/microsoft/CodeBERT

Another derivate is CodeBERT-MLM. This MLM version is available on Hugging Face[3], and allows the replacement of predefined mask tokens in the input with appropriate source code.

For this thesis, only the quality estimation task of CodeReviewer/CodeBERT and the MLM are of relevance.

To classify a given input sequence and counterfactual candidates, the quality estimation task of the CodeReviewer model is used. The model returns an array containing two values, which are the confidences in the classes determined by its index. I.e., index 0 contains the confidence the model has in 0 being the appropriate classification for the given input sequence. The classification is established by determining the larger of the two values. The `classify(self, source_code: str)` function returns a tuple of the classification and the confidence of the model in said classification. Therefore, CodeReviewer can be used in both GS and kEES algorithms.

According to the authors of [26], CodeReviewer should achieve an accuracy of between 74.04% for Java and 82.7% for Ruby, and an F1 score of 70.53 and 89.23, respectively for the code quality estimation task. However, the fine tuned checkpoints are not publicly available. Therefore, for this thesis, an attempt was made to fine tune the base CodeReviewer model available on Hugging Face. For this purpose, the training and testing data is published and available for download[4], and the GitHub repository provides both training and testing scripts[5]. After running the `train-cls.sh` script for 36,000 iterations, an accuracy of 68.78% and an F1 score of 68.37 was achieved according to the test script `test-cls.sh`. After this, both the accuracy and F1 score only declined when training for further iterations. However, it was not possible to reproduce this accuracy on any data other than the training and test data. Instead, the fine tuned model classifies every source code given to it as "1", which is the class for vulnerable source code. This also applies to models on Hugging Face fine tuned by third parties[6].

Another fine tuned version is available on GitHub[7]. This variant is trained for the paper Natural Attack for Pre-trained Models of Code by Yang et al. [48], which searches for ways to find adversarial samples which still look natural to humans. The fine tuned models was provided by the authors of the mentioned papers for this thesis and is the model used in this thesis.

4.1.2 PLBart

Another model trained on source code is PLBart [2], which is available on Hugging Face[8]. The authors of PLBart also provide a fine tuned version of the model for code

[3] https://huggingface.co/microsoft/codebert-base-mlm
[4] https://zenodo.org/records/6900648
[5] https://github.com/microsoft/CodeBERT/tree/master/CodeReviewer/code
[6] https://huggingface.co/mrm8488/codebert-base-finetuned-detect-insecure-code
[7] https://github.com/soarsmu/attack-pretrain-models-of-code/tree/main/CodeXGLUE/Defect-detectionattack
[8] https://huggingface.co/uclanlp/plbart-base

classification tasks. These tasks are clone detection and vulnerability detection, where the latter task is of interest for this thesis. The vulnerability detection task was implemented to test how the model performs on previously unseen programming languages, and is therefore only available for C++ code. PLBart claims an accuracy of 63.18% on the vulnerability detection task on the CodeXGLUE benchmark [29]. A number of fine tuned variations of the base PLBart model are published on Hugging Face, including the `uclanlp/plbart-c-cpp-defect-detection` model[9] used in this thesis.

As the model outputs both a classification and a confidence score, it can be used with both brute force algorithms such as the *k*EES and heuristic algorithms such as the GS. A Python code sample on how to load the model and compute a classification and confidence score for a given input string is given in Listing 4.1. The model provided on Hugging Face is already fine tuned on the vulnerability detection task. However, the required training and testing scripts, and scripts to download training and testing data to fine tune the base model are provided on GitHub[10] anyway. This script outputs a classification of 1 for vulnerable source code, and 0 for secure and or invulnerable source code.

```
1  path = "uclanlp/plbart-c-cpp-defect-detection"
2  model = PLBartForSequenceClassification.from_pretrained(path)
3  tokenizer = AutoTokenizer.from_pretrained(path)
4
5  def classify(self, source_code: str) -> (bool, float):
6      inputs = self.tokenizer(source_code, return_tensors="pt",
           truncation=True)
7      input_ids = inputs["input_ids"].to(self.device)
8      attention_mask = inputs["attention_mask"].to(self.device)
9      with torch.no_grad():
10         logits = self.model(input_ids=input_ids,
               attention_mask=attention_mask).logits
11         clazz = logits.argmax().item()
12         return int(clazz) == 0, float(logits[0][clazz])
```

Listing 4.1: The Python souce code needed to instantiate the PLBart model and to get a quality estimate for a given source code snippet.

4.1.3 VulBERTa

VulBERTa [20] is a model aimed specifically at detecting vulnerabilities in C/C++ source code. To this end, it is trained on 1,101,075 functions from 1060 open source GitHub repositories and an additional 1,274,366 functions from the Draper data set introduced in [39]. Furthermore, VulBERTa profits from a tokenisation pipeline which is tailored towards C/C++ code using Clang[11]. The resulting tokenizer allows the preservation of the

[9] https://huggingface.co/uclanlp/plbart-c-cpp-defect-detection
[10] https://github.com/wasiahmad/PLBART
[11] https://clang.llvm.org/

syntactic structure. This holds true even after it is converted into a token sequence. This sequence of tokens gets further processed to account for predefined tokens. Predefined tokens include tokens reserved by the VulBERTa transformer, such as `<pad>` or `<mask>`, C/C++ keywords and punctuation such as `int`, `void` or `++`, and C/C++ API calls such as `strlen` or `memcpy`.

Two versions of VulBERTa exist: VulBERTa-MLP, where the last stage of the transformer is a Multi Layer Perceptron (MLP), and VulBERTa-CNN, with a Convolutional Neural Network (CNN) as last stage. This thesis uses the former one, as it has been made available on Hugging Face by a third party[12] and it performs slightly better. VulBERTa-MLP achieves an accuracy of 64.75% on the CodeXGLUE [29] benchmark, which at the time [20] was written, was the third best score after CoText and C-BERT, while only having 55.05% of the number of model parameters of CoText. On the D2A [51] benchmark, VulBERTa-MLP scored the first place with an accuracy of 62.3% and VulBERTa-CNN scored second place with an accuracy of 60.68%. Both model variants beat C-BERT in this benchmark.

As the model returns both a classification and a confidence score, it can be used in both *k*EES and GS algorithms. Note, that as the model uses Clang in its tokenisation pipeline, Clang for Python must be installed, and the path to the library file must be set to the `libclang.dll` file. The model outputs a classification of `"LABEL_1"` for vulnerable source code, and `"LABEL_0"` for secure and or invulnerable source code.

4.1.4 CodeT5

The CodeT5 model [47] is based on the Text-To-Text Transfer Transformer (T5) model [37]. Whereas the original T5 model is only trained on natural language, the CodeT5 model is also trained on programming languages and on converting natural language into programming language and vice versa. The architecture of CodeT5 allows the model to use the names of identifiers to extract further information out of source code, which helps by giving the model additional context. E.g. a function name `binarySearch` may already give insight in what this function will do. The model can be fine tuned on a number of different tasks, including code summarization, code refinement, and defect detection. The latter task makes this model a candidate for this thesis. According to the paper [47], CodeT5 achieves an accuracy of 65.78% on the CodeXGLUE benchmark [29], thereby outperforming both PLBart (Section 4.1.2) and VulBERTa (Section 4.1.3). This thesis uses a version of the CodeT5 model that is finetuned on the defect detection task and is available on Hugging Face[13]. As with other models trained for the CodeXGLUE benchmark, the model outputs a classification of 1 for vulnerable source code, and 0 for secure and or invulnerable source code.

[12] https://huggingface.co/claudios/VulBERTa-MLP-Devign
[13] https://huggingface.co/mcanoglu/Salesforce-codet5p-770m-finetuned-defect-detection

4.1.5 Version Incompatibilities

As is common to many Python libraries and open source code [19][21][49], compatibility issues exist with the models listed in this section. There is no version of the Hugging Face transformers library[14] which works with all models and all search algorithms. Instead, different use cases require different versions, as listed in Table 4.1.

Search Algorithm	(Graph) CodeBert	PLBart	VulBERTa-MLP	CodeT5
GS	$\geq$ 4.17.0	$\geq$ 4.17.0	$\geq$ 4.37.0	$\geq$ 4.37.0
Greedy Search	$\geq$ 4.17.0	$\geq$ 4.17.0	$\geq$ 4.37.0	$\geq$ 4.37.0
kEES	$\geq$ 4.17.0	$\geq$ 4.17.0	$\geq$ 4.37.0	$\geq$ 4.37.0
LIG	4.17.0 - 4.22.0	4.17.0 - 4.22.0	$\geq$ 4.37.0	-

Table 4.1: The transformer versions required to run search algorithms on the given models. Note that these versions are not necessarily the earliest or latest versions for which the current use case will work, but these versions are tested and verified to function.

4.2 Methodology

This section describes the methodology followed to create and evaluate the results for this thesis.

4.2.1 Test data

The evaluation of the performance of the system to search for counterfactuals uses data from the CodeXGLUE benchmark [29], which is available on GitHub[15], which uses the Devign dataset [52]. This test data consists of C++ source code with vulnerabilities that could be exploited to attack software. Vulnerabilities include resource leaks, use-after-free and other bugs and insecure source code. The source code originates from the open source projects FFmpeg and QEMU. Overall, the first 44 to 95 source code samples that are labeled as vulnerable and lie in the validation split of the dataset are taken. The exact number of investigated source code samples depends on the search configuration.

4.2.2 Hardware

All experiments were performed on the Idefix server of the TU Wien. Idefix has an AMD EPYC 7452 32-Core Processor CPU with 1007 GB RAM. To boost NN performance, it contains four NVIDIA GeForce RTX 3090 GPUs. Experiments were sped up by running the system for finding counterfactuals in four separate Python processes in parallel, each one using its own GPU.

[14] https://huggingface.co/docs/transformers/en/index
[15] https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/Defect-detection/dataset

4.2.3 Code formatting

The models used to classify source code were trained at least partially on source code taken from real world repositories, which are expected to at least in part adhere to formatting guidelines for their specific programming languages. However, the Tokenizer alone as described in Section 3.2.1 cannot guarantee a certain output formatting, especially not one that follows the guideline for a specific language. Therefore, the intermediary output of the Tokenizer is passed to a formatting script, such that the final output does adhere to formatting guidelines. This script performs some preprocessing steps on its input and then spawns a `clang-format`[16] subprocess, which is given the preprocessed source code and returns it formatted. `clang-format` ignores any syntax errors or missing dependencies, which may easily be introduced by the perturbations and formats code on a best effort basis, even if it is erroneous. This helps with giving the model source code which is as close to its training data as possible, even if counterfactual candidates are not syntactically correct.

4.2.4 Error handling during the search

All components provide some degree of error handling in case an error occurs during the search for counterfactuals. The most common source for errors are too long inputs, which cannot be handled by the models. In this case, the model specific tokenizers are instructed to truncate the input to an appropriate length for the models before the search starts. It is possible that this truncation removes the part which the model considers defective, which would mean that the model classifies a previously defective input as invulnerable. Then, the search is aborted before it is started and an `InvalidClassificationResult` is returned. Therefore, the search should only ever be started with inputs of appropriate length. During the search, the length of the input could increase again, e.g. through source code generated by a MLM Unmasker. This increase could lead to too long inputs once more, so even during the search the model specific tokenizers are tasked to truncate inputs. However, in some case they fail to do so, and the model throws an error. The search algorithms then catch this error or other errors and abort the search for the affected branch, but continue on other branches. To prevent endless looping over the same best counterfactual candidate whose descendants throw errors in the Greedy search, each candidate is associated with an `error_count` variable. This variable gets incremented every time a direct descendent of a candidate results in an error. Once it reaches a value of three, the parent candidate is removed from the pool. One exception to the error recovery is the LIG search. Since this search algorithm only ever iterates over the results of the previous search iteration, there is no second branch which could be continued. Therefore, the search is completely aborted without the chance to retrieve a counterfactual. Truncating the source code input leads to information loss [27] and might make it impossible for a model to find vulnerabilities or it could introduce new vulnerabilities, but it is necessary to make the models work at all. Another option would be to split the source code into subsequences of appropriate length. However, this would

[16] https://clang.llvm.org/docs/ClangFormat.html

not solve the issue, as now a vulnerability could be spread over several substrings, and with the context of the previous and following substrings missing, might not even be a vulnerability on its own. An `InvalidClassificationResult` is also returned if the model erroneously classifies the original input as invulnerable in the very beginning of the search.

4.2.5 Removed token classification

To give further insight into which removed tokens resulted in counterfactuals, Clang[17] for Python is used to parse the removed tokens. This parsing results in the following categories.

- `FUNC_DEF`: This category corresponds to the definition of functions. In the context of this evaluation, a function definition is any number greater than zero of keywords, followed by an identifier, followed by an opening round bracket. E.g.: `inline static long int foo(int a, int b)`
- `LOOP`: this category includes `for`, `while`, and `do-while` loops. E.g.: `for(i = 0; i < 10; i++)`
- `KEYWORD`: this category includes all keywords, that are not covered by the `LOOP` category. E.g.: `char32_t count;`, `return 0;`, `if (i==0)`
- `FUNC_CALL`: function calls are identifiers followed by an opening round bracket. E.g.: `c = foo(a,b);`
- `PUNCTUATION`: All punctuation characters such as brackets, dots, semicolons and others.
- `IDENTIFIER`: All tokens which Clang interprets as identifiers in the context of the given source code.
- `LITERAL`: All tokens which Clang interprets as literals in the context of the given source code.
- `UNKNOWN`: Everything else.

Note that the tested string does not need to be syntactically correct and that the categories are presented in the order in which the input string is tested. This means that the input is firstly checked whether it is a function definition, and only if this test does not terminate positively, further tests are executed. Each token can only be attributed to one category. If a given input satisfies several definitions, only the first matching one is reported.

[17] https://clang.llvm.org/

4.3 Measurements

To compare the individual components and resulting search configurations, this thesis takes an array of measurements. The results of these measurements are discussed in the corresponding subsections in Section 4.4, and in Section 6.

4.3.1 Invalid Classifications

A classification is considered invalid, if a source code sample is labelled as vulnerable in the CodeXGLUE dataset, but the employed classifier classifies it as invulnerable. In this case, a search for a counterfactual is useless, as the model is not able to correctly identify the vulnerability in the sample. A high rate of invalid classifications indicates a poorly performing model of code.

4.3.2 Number of Changes for a Counterfactual

As explained in Section 1, counterfactuals that are closer to the original input are preferable. For this thesis, one such evaluation is the number of tokens that has been changed to achieve a counterfactual. To normalize this data for the varying source code snippets in the data sets, this value is given as the number of changes relative to the number of tokens. A source code snippet with e.g. 15 tokens producing a counterfactual with 18 tokens would result in a score of 120%.

4.3.3 Search Performance per Search Algorithm

The selected search algorithm has a great influence on the size and structure of the resulting search space. To measure this influence, the time per counterfactual and per input token is investigated, as well as the number of counterfactuals per input token. Search algorithms that find efficient ways to navigate the search space will speed up the search and deliver more counterfactuals.

4.3.4 Search Performance per Model of Code

To investigate the influence on the search space of the models of code, similar measurements as for the previous subsection are taken. The model of code can influence the search by providing a more accurate and stable confidence score in its classification, and by correctly classifying invulnerable inputs as invulnerable. Furthermore, faster models of code will speed up the search.

4.3.5 First Counterfactual Search Performance

For this measurement, the time needed to find only the first counterfactual is considered. If no counterfactual can be found, the search run is ignored. This measurement might be important in real world applications, where a human user would probably not want

to wait until the whole search is finished, but might be content with the first or second counterfactual delivered.

4.3.6 LIG Search

As the LIG search does not use the same Tokenizers, Unmaskers and Perturbers as the other search algorithm, their result cannot be directly compared. This is because of the internal strucutre of the LIG search algorithm and the underlying Captum library. Instead, only the number of counterfactuals per search run and the time per counterfactuals can accurately be compared against each other. This also means that if a counterfactual would require the addition or replacement of a token, it will never be found using the LIG search.

4.3.7 String Similarities

As discussed in Section 1, counterfactuals that are closer to the original input are of higher value. Another option that that explained in Section 4.3.2, is to quantise this closeness by calculating the string similarity of a counterfactual to its input. This closeness is computed using the Jaro distance, as explained in Section 2.5. The actual values of the similarity score is not relevant, but the values of the different search configurations can be compared with each other. The higher a value is, the closer the counterfactual is to the original input.

4.4 Results

This section displays the results of the developed system, as discussed in Section 4.3. Due to the high number of different search configurations, the full results are split over several tables and figures, and are abbreviated as shown in Table 7.1. The full data is given in Appendix 7, with the relevant parts highlighted here. The parameters for each configuration are presented in Table 7.2. Since not all parameters are applicable to each configuration, some cells in this table are marked as "not applicable" or "n.a.". Performance metrics are displayed in Table 7.3. The values for the parameters were chosen empirically to give a compromise between search duration, to allow for a large number of search runs, and a high chance to find counterfactuals within a search run. Table 7.4 gives insight into the types of removed tokens that resulted in counterfactuals with less than three changes to the input. Counterfactuals with more than two changes to the input are assumed to be too far off the input to make such an evaluation meaningful. The distribution of the relative number of changes to get a counterfactual with respect to the input size is shown in Fig. 4.1.

4.4.1 Invalid Classifications

Due to the inaccuracies of the used models of code, some code samples labeled as vulnerable in the CodeXGLUE dataset are erroneously classified as invulnerable. As discussed in

Section 4.2.4, search runs affected by this issue return an `InvalidClassificationResult`. The absolute and relative numbers of these occurrences, as well as the total number of source code snippets evaluated, are given in Table 4.2.

Classifier	Total source code samples	Invalid classifications	Percentage of total [%]
CodeBert	45	20	44.44
CodeT5	50	22	44.0
PLBart	44	19	43.18
VulBERTa-MLP	95	45	47.37

Table 4.2: The numbers of total classifications and invalid classifications for each classifier. An invalid classification is when the model identifies a source code sample as invulnerable when it is labeled as vulnerable in the CodeXGLUE test dataset. In this case the search for counterfactuals is aborted before it is started. The fourth column gives the relative percentage with respect to the total number of evaluated source code snippets.

4.4.2 Number of Changes for a Counterfactual

Figure 4.1 shows the distribution of changes to the original input measured against the size of the input. E.g. if an input input consists of 20 tokens and is classified as a counterfactual after five changes, 25% of the input were changed to produce this counterfactual. For simplicity reasons, the system does not treat reoccurring changes differently, i.e. if the same token gets changed multiple times, this counts as multiple changes, even if previous changes are lost. Counter intuitively, the number of counterfactuals does not increase with more changes to the input. This might be because after many changes the source code will not only be too far off the original input, but also far off correct C++ source code, and therefore not invulnerable according to the models of code. Similar data is presented in Figure 4.2, where it is displayed by search configuration.

4.4.3 Search Performance per Search Algorithm

The choice of the search configuration greatly impacts the search performance. As expected, the exponential nature of kEES means that the average processing time per input token is highest for this search algorithm, as displayed in Figure 4.3. However, the large search space also means that kEES produces the most counterfactuals per input token as shown in Figure 4.5. Furthermore, as it uses a k of two, all its counterfactuals are relatively close to the original input and should therefore be the most useful. Even though kEES produces a large number of counterfactuals, it is the slowest algorithm per counterfactual for all search configurations as depicted in Figure 4.4.

Figures 4.3 and Figure 4.4 suggest that the unmasking operation of the CodeBertUnmasker introduces a significant time overhead which slows down the search as opposed to the MutationPerturber and RemoveTokenPerturber. Between the latter two perturbers, the

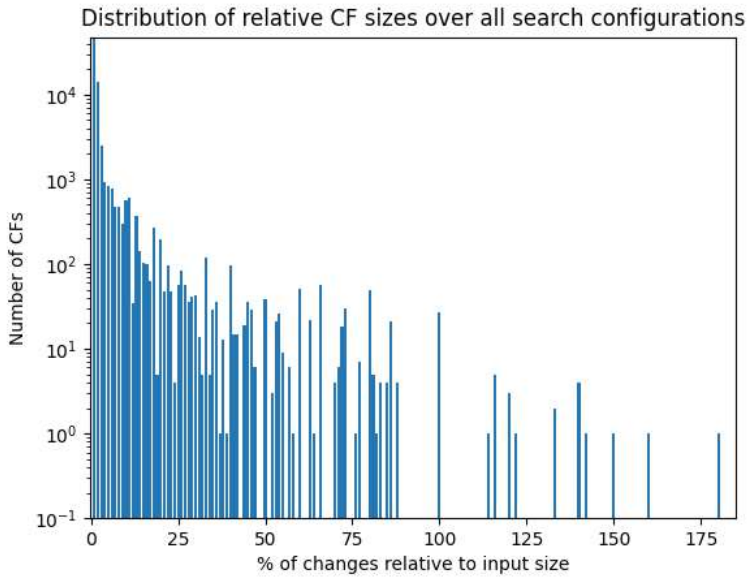

Figure 4.1: The percentage of changes to produce a counterfactual with respect to the number of input tokens. If the same input token is changed more than one time, this counts as multiple changes. This graphics shows the numbers over all search configurations.

RemoveTokenPerturber is faster on a per token basis, but slower when measured per counterfactual than the MutationPerturber.

4.4.4 Search Performance per Model of Code

Different models also influence the search performance differently. Figure 4.6 shows how long each model takes to find a counterfactual on average for each search algorithm. Note that since VulBERTa-MLP did not find any counterfactuals, there is no data for it present in the figures. As shown in Figure Figure 4.7, CodeBert is on average the fastest model for every search algorithm per input token, but the slowest per counterfactual. This can be explained by the numbers in Figure 4.8, which show that CodeBert delivers the fewest counterfactuals per token.

Overall, the data shows that kEES is the slowest search algorithm by a substantial margin. Both heuristic search algorithms greatly outperformed kEES. Depending on the exact search configuration, the GS or greedy search might be faster. The slowest of all search algorithms is the LIG search. However, its search duration is not directly comparable to the other algorithms, as it terminates its search after the first counterfactual is found. Furthermore, depending on the parameters, it needs to perform costly computations after each removed token. When it comes to finding the most counterfactuals in some time frame, CodeT5 is slightly slower than PLBart, which in turn is considerably faster than CodeBert. On a per token basis, those three models are similarly fast, with CodeBert

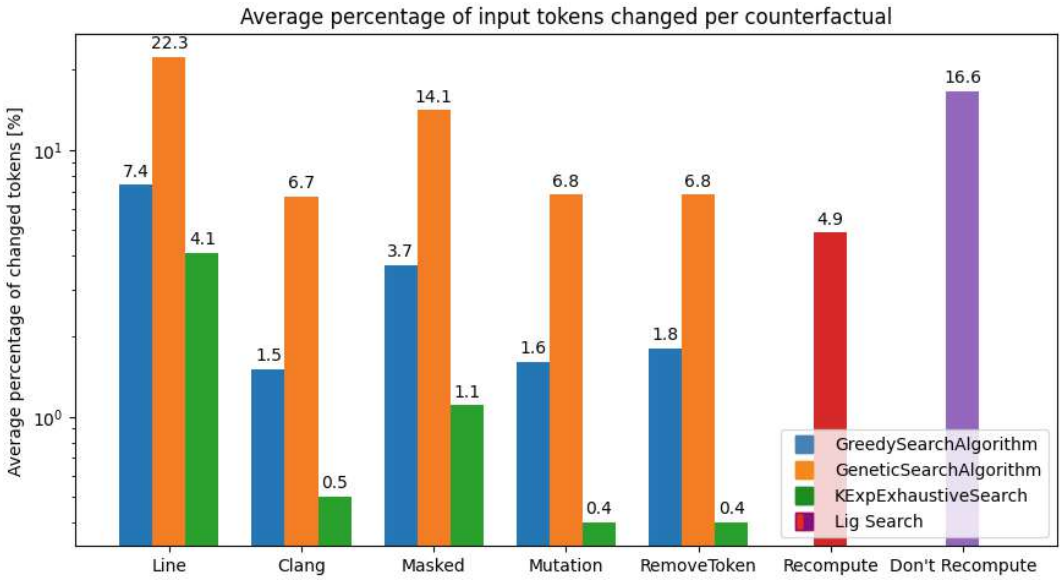

Figure 4.2: The percentage of changes to produce a counterfactual with respect to the number of input tokens by search algorithm.

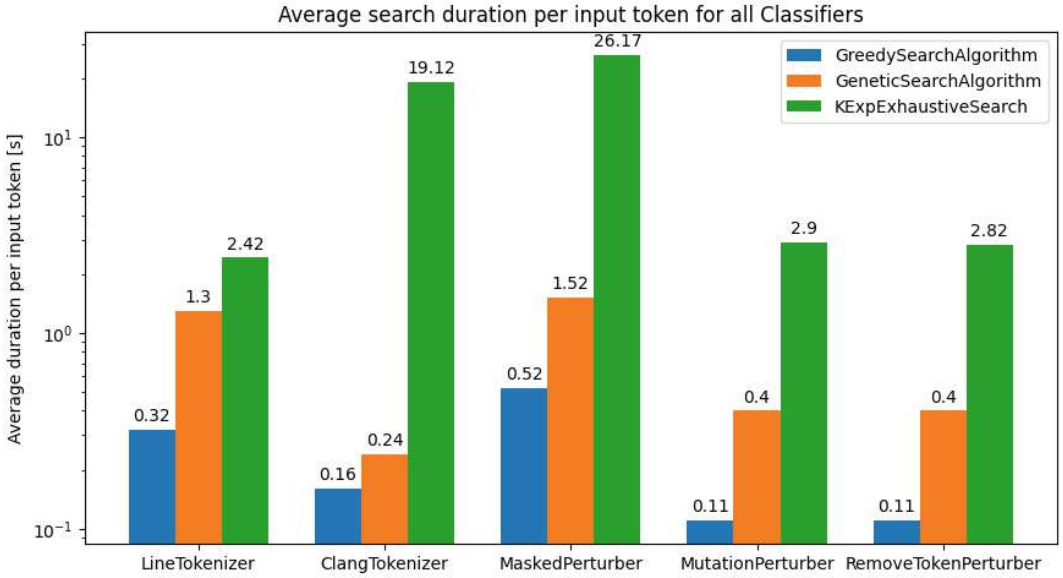

Figure 4.3: The average processing time per input token over all search configurations for each search algorithm.

being the fastest by a small margin.

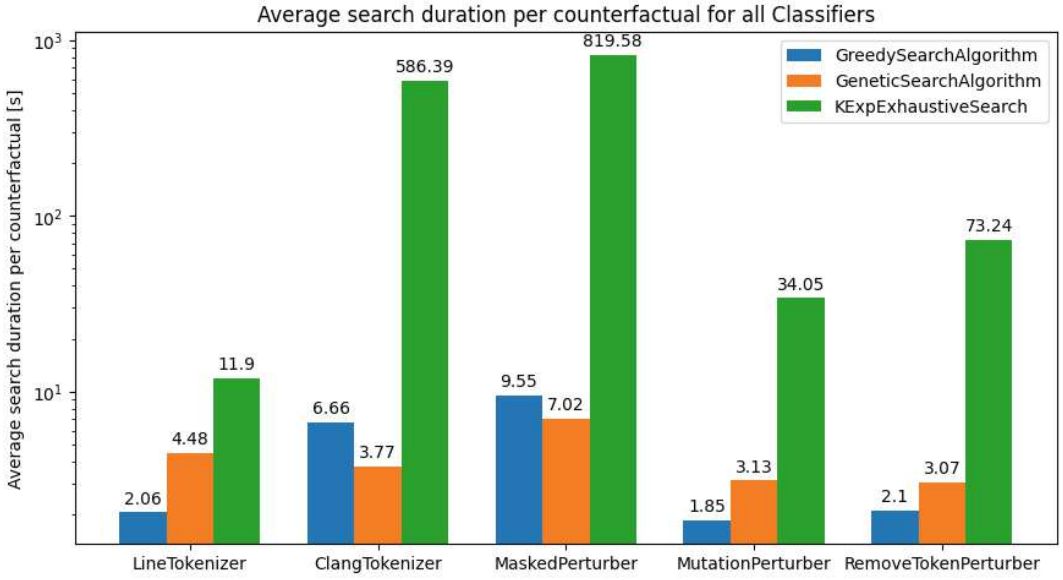

Figure 4.4: The average processing time to find a counterfactual over all search configurations for each search algorithm.

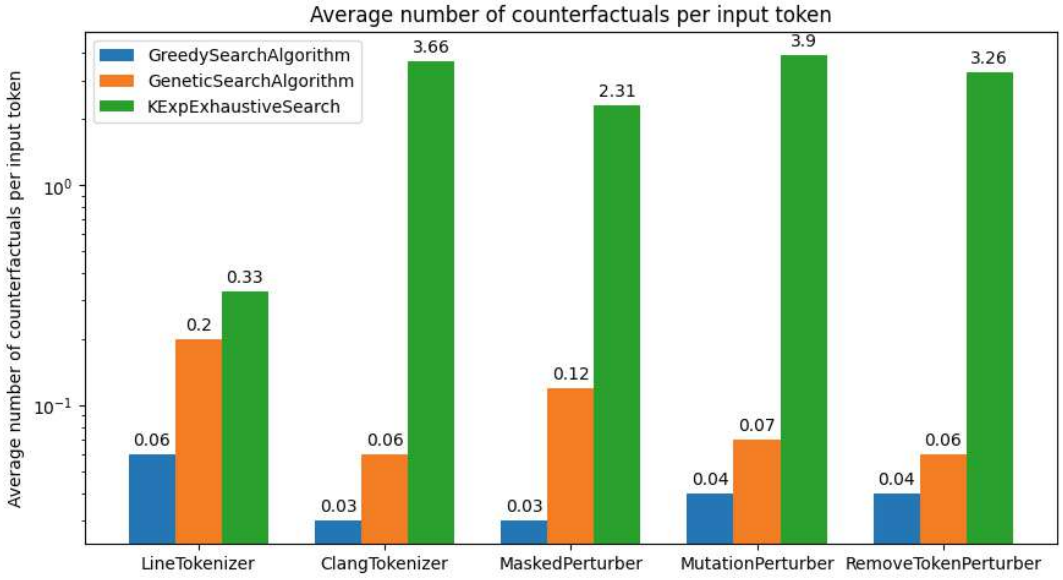

Figure 4.5: The average number of counterfactuals produced by each search configuration for each search algorithm per input token.

4.4.5 First Counterfactual Search Performance

The time needed for each search configuration to produce its first counterfactual is displayed in Figure 4.9. For this comparison, the greedy search is consistently able to

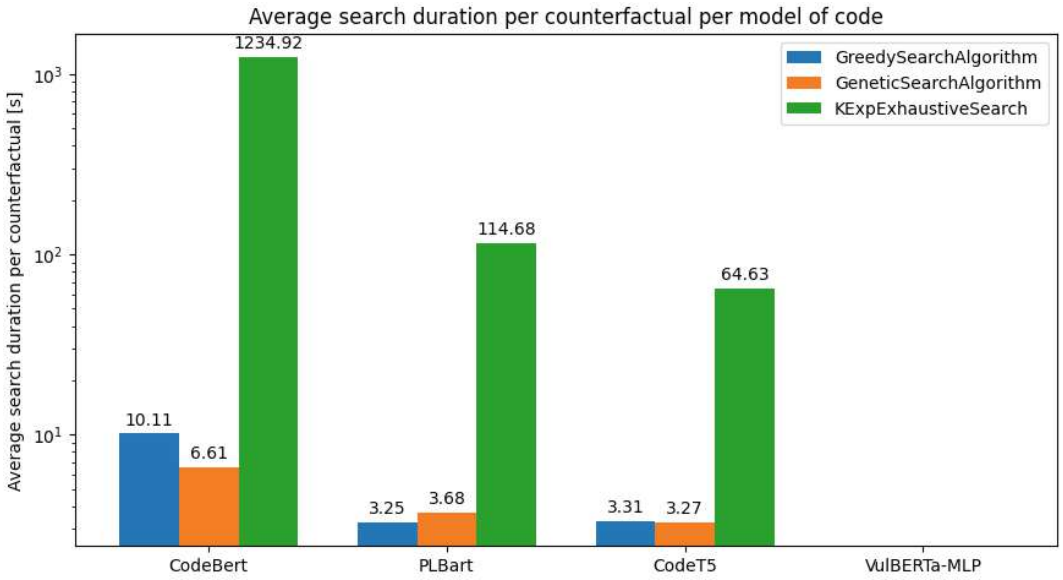

Figure 4.6: The average search duration per counterfactual for each model of code.

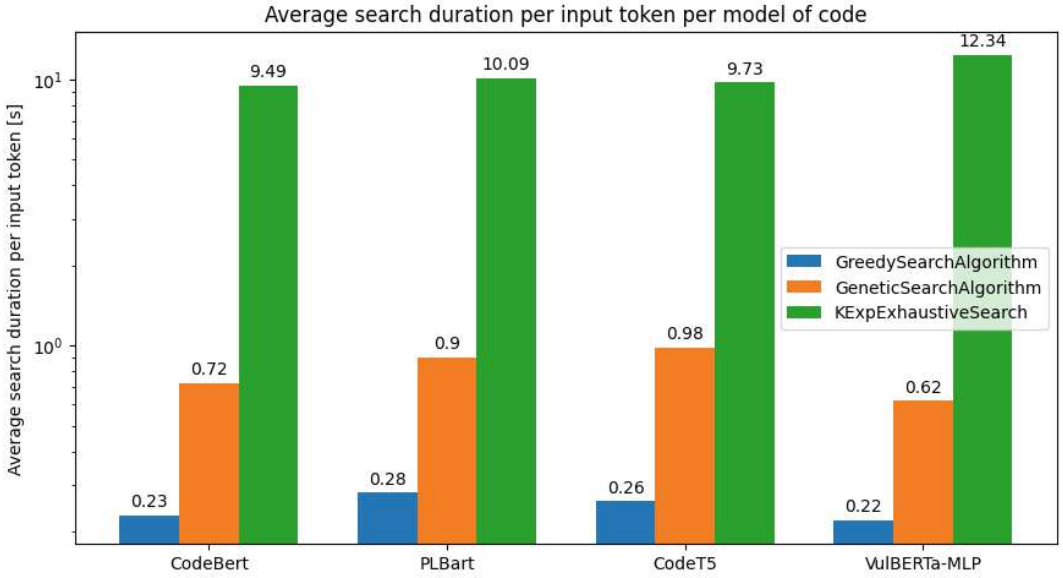

Figure 4.7: The average processing time to find a counterfactual over all models of code for each search algorithm per input token.

outperform all other search algorithms.

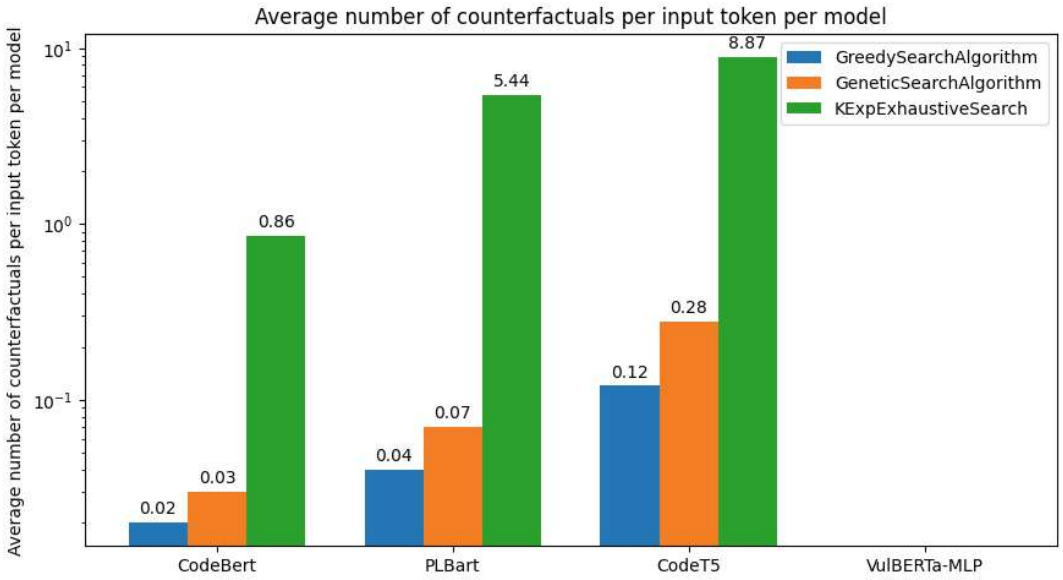

Figure 4.8: The average number of counterfactuals produced per input token per model. Note that VulBERTa-MLP did not produce a counterfactual.

4.4.6 LIG Search

As the LIG search does not use the same Tokenizers, Unmaskers and Perturbers as the other search algorithm, their result cannot be directly compared. Instead, only the number of counterfactuals per search run and the time per counterfactuals can accurately be compared against each other. This is done in Figure 4.10, which displays the average number of counterfactuals produced per search run, and Figure 4.11, which shows the average time to find a counterfactual for all search algorithms.

These figures show that the LIG search produces counterfactuals for less inputs than other search algorithms, and it finds them slower than other search algorithms. This is because the LIG search only ever removes tokens from the input.

4.4.7 String Similarities

Figure 4.12 shows the results of the comparison Jaro distances of counterfactuals and their original inputs. No search configuration is significantly better than any other. However, the greedy search performs consistently worse than the other search algorithms for all search configurations. The LIG search is at best as good as the greedy search, at worst it is worse than it. The full data containing all string similarities is given in Table 7.3.

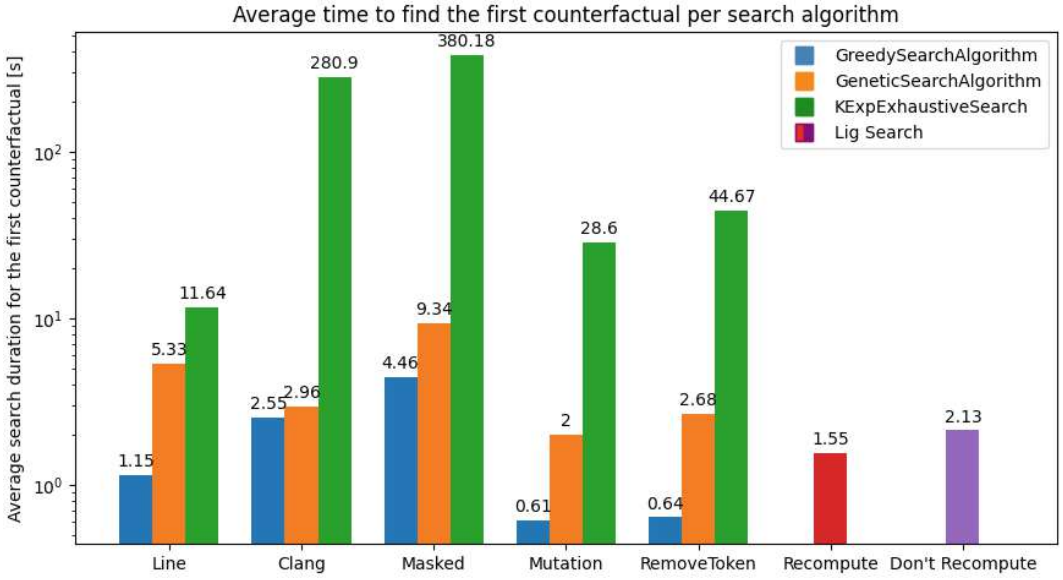

Figure 4.9: The average time to find the first counterfactual over all search configurations for each search algorithm. "Recompute" and "Don't Recompute" indicates the state of the `recompute_attributions_for_each_iteration` parameter, i.e. if the LIG attributes should be recomputed after each removed token. Note that the LIG search does not use Perturbers or Tokenizers the way the other search algorithms do. This data only includes times from search runs were at least one counterfactual was found, hence the disparity with Figure 4.11 for the LIG search.

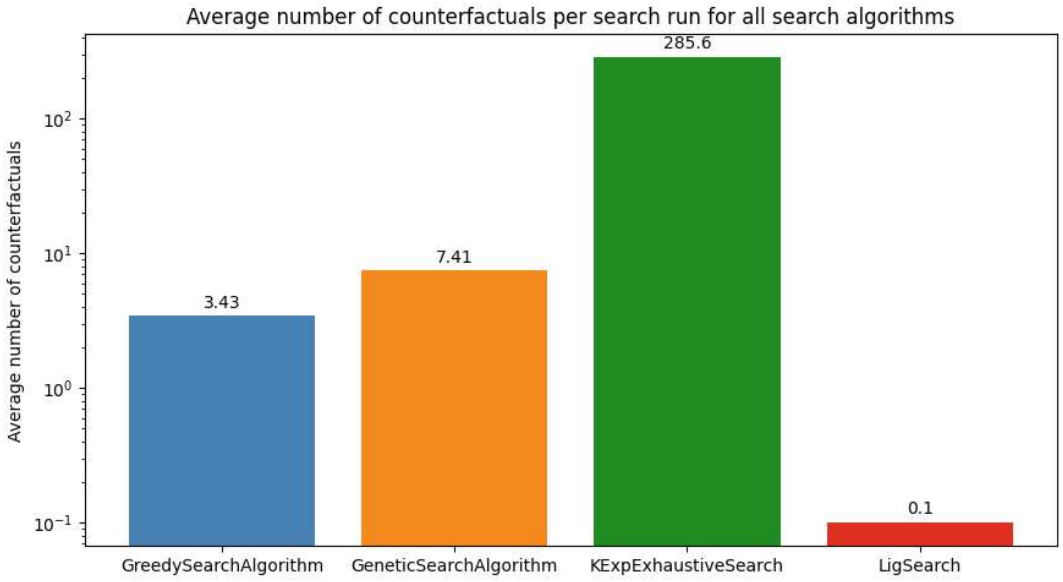

Figure 4.10: The average number of counterfactuals produced per search run for all search algorithms, including the LIG search. As the LIG search produces at most one counterfactual, this data shows that it found a counterfactual on 10% of search runs.

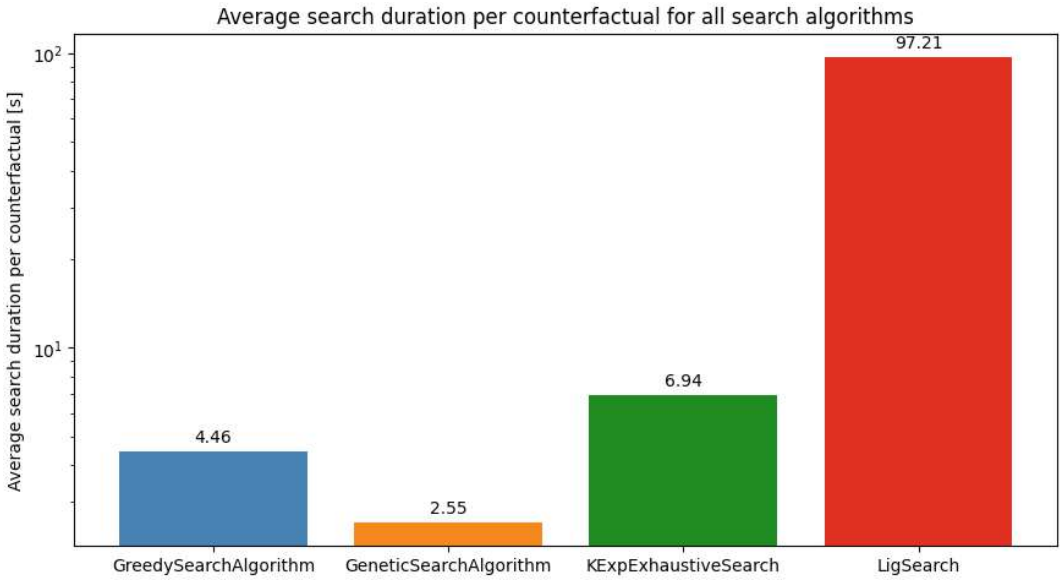

Figure 4.11: The average time to produce a counterfactual for all search algorithms, including the LIG search. These times also include search runs in which no counterfactuals were found, hence the disparity with Figure4.9 on the data for the LIG search.

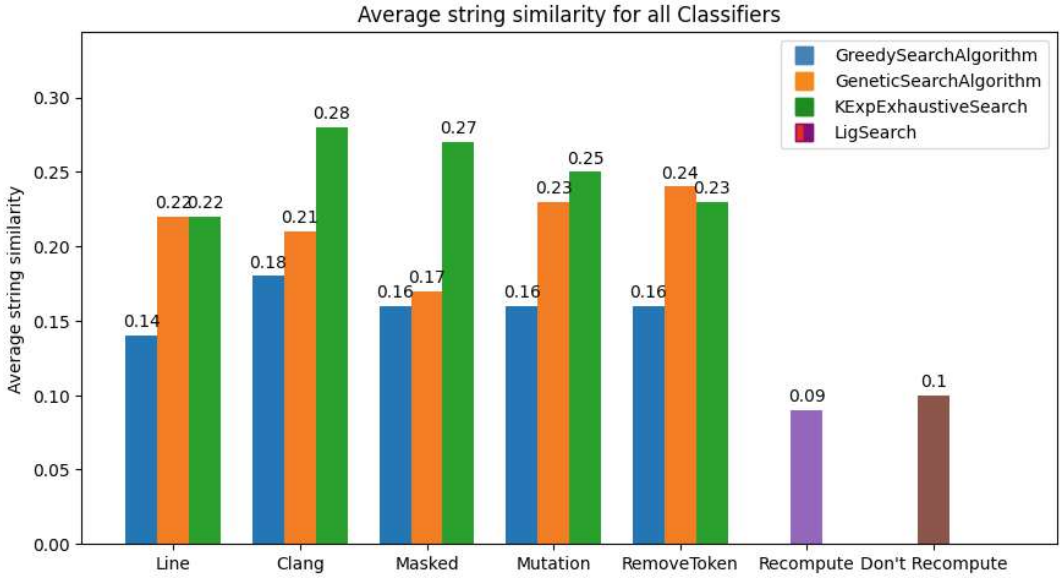

Figure 4.12: The average string similarity from each produced counterfactual to the original input. This similarity is calculated using the Jaro distance, as explained in Section 2.5. Higher values mean higher similarities to the input and are therefore better. "Recompute" and "Don't Recompute" indicates the state of the `recompute_attributions_for_each_iteration` parameter, i.e. if the LIG attributes should be recomputed after each removed token.

CHAPTER 5

Related Work

Humans tend to place less trust in the answers of block box systems if they are not provided with a reason as to why this output was generated [15]. Therefore, together with the emergence of black box NNs, ways to make them explainable were developed [17]. Another use case for understanding why outputs occur from specific inputs is to use this knowledge as an attack vector [16]. This allows to change the classification of an image by altering just a single pixel in the image [43] or applying noise to an image, which may be undetectable by humans [13]. However, such adversarial attacks are not the same as generating counterfactuals, as the aim of an attack is not to provide knowledge about the reasoning of the model, but to get a different output on what appears to be the same input. Here, no emphasis is put on making the differences actionable or plausible. Nevertheless, similar strategies may be followed when searching for counterfactuals or adversarial examples [17]. For these reasons, this thesis will not go into detail about generators if their sole purpose is to create adversarial examples. Most existing model of code based strategies of finding counterfactuals can be subdivided into four distinct categories [17]:

- **Optimization:** An optimization strategy defines some loss function and uses optimization algorithms to minimise the loss according to this function. A notable example is proposed by Wachter et al., which is one of the first counterfactual explainer using this strategy. It can use the Adam optimizer [24] on differentiable models, such as NNs, to approach the least possible loss [46].

- **Heuristic Search Strategy:** Here, heuristics are used to minimize some cost function at each iteration. This class of explainers is typically more efficient than optimization explainers, but the produced results are not always optimal. Published in 2014, [31] is another early model agnostic explainer. It uses a best-first search with pruning by following local improvements. The aim of the algorithm is to

find a minimal set of words, which, when removed from a document, changes the classification of the text from some classifier.

- **Instance-Based:** This strategy requires a pregenerated dataset of potential counterfactuals. Counterfactuals are found by choosing entries within this dataset which have the highest similarity to the input. For each counterfactual, [40] searches the dataset for the closest sample to the input with a different label than the input. This has the advantage that the counterfactual will always be plausible, as all counterfactual candidates do actually exist in the dataset. A disadvantage of this search is the high computational cost with calculating the distance from the sample to all entries in the data set. Somewhat overcoming this disadvantage is possible by limiting the search to a smaller subset of the dataset, but this also means that the optimal counterfactual might not be present in this subset.

- **Decision Tree:** A decision tree is trained to closely approximate the behaviour of the model of code. Then, the tree structure, which is inherently understandable, is used to find counterfactuals. As described in [18], such a decision tree can either be trained on the same data set as the model to explain, a subset of this data set, or a different, synthetic data set. Furthermore, the tree structure allows to generate high quality counterfactuals. Following only paths that are actionable at each tree node guarantees actionability. Even if paths have to be followed that are not actionable, those properties of the counterfactual can be overwritten with the corresponding properties from the input. A minimal distance to the original input sample can be achieved by choosing paths that are more similar to the input than others. Before a counterfactual candidate can be presented as such, it must undergo validation, as the mentioned correction steps might result in an invalid candidate, i.e. in a candidate that receives the same classification as the input.

Furthermore, explainers can be classified as Model Specific, if they only work on a specific model, or Model Agnostic, if they can be used on any model. Additional distinctions can be made concerning the data type the explainer can work with, e.g. text, source code, images, and others.

Models of code, such as Microsoft CodeReviewer [26], attempt to tell if some source code may contain bugs or have performance or security issues. However, they cannot explain their reasoning, and generally only provide a binary output classification, i.e. defective or not defective, and a confidence score. Cito et al. [8] specifically aims at producing counterfactuals for such source code quality estimation models. [8] produces a set of tokens and replacements for these tokens, such that each replacement changes the classification of the code input. The algorithm attempts to find counterfactuals of increasing number of contained tokens, by adding a replacement mapping to the most promising counterfactual candidate of a set of candidates. Then, the candidate is tested whether it is classified differently than the original input. If so, a new counterfactual has been found. Replacements tokens are generated by iterating over the tokens in the

input and substituting each token with a mask token. Feeding the resulting source code snippet to a MLM, gives a set of possible replacement tokens for this mask. This set of tokens is evaluated for which replacement results in a counterfactual. Users of this system found the generated explanations useful or very useful 83.3% of cases. The authors note that the runtime of up to 10 minutes on large inputs somewhat reduces the usefulness of the algorithm in real world applications, where developers would want to have near instantaneous feedback when committing source code. As the algorithm makes a local decision at each iteration, it can be classified as a Heuristic Search Strategy. It is not tied to a specific model and is therefore Model Agnostic.

A more specialized counterfactual search strategy is [28]. This system does not use black box models of code, but still explores a search space which is spun up through mutating a source code snippet, and only works for a specific programming language. Firstly, a source code snippet is evaluated against a predefined test suite, which it initially does not pass completely. This test suite contains both positive and negative tests, to ensure that both improvements and regressions can be identified. [28] then runs the tests against the source code snippet, with an error localizer supervising the test runs. This error localization injects call backs before each statement in the source code snippet to track the progress and runtime performance of the execution. Potential counterfactuals are generated by applying transformations to program statements that led to errors. These transformations include additional conditions to if statements, additional conditional statements which execute the original statement if the condition is satisfied, or add `break`, `continue`, or `goto` statements in case they are satisfied. Further transformations include memory initialization, value replacements and copy transformations. Conditions for the transformations are generated from comparisons of local or global variables and values that occur somewhere in the program in conjunction with the selected variable. [28] runs the transformed source code snippets through the test suite, and reports all improvements.

CHAPTER 6

Conclusion

This chapter gives a short overview of the findings of this thesis and possible future research.

6.1 Summary

For this thesis, a program was developed that uses an array of different search algorithms, perturbation functions, and Tokenizers to explore how these individual components contribute to the search for counterfactuals.

- **RQ1**: What is the search space for possible perturbations with respect to different search heuristics?

  With the exception of the LIG search, which only finds counterfactuals in 10% of search runs, kEES is the slowest search algorithm by every measure. This shows, that guiding the search through some heuristic, and thereby focusing the search on promising areas of the search space, can greatly benefit the search for counterfactuals. Furthermore, kEES also takes the longest to find its first counterfactual. This is expected since kEES always starts by perturbing the first token in the source code sample, which often contains some parts of a function header, such as the name or visibility, which most likely is not part of some vulnerability. This shows that some understanding of the structure of the problem is helpful.

  Figure 4.2 shows how many changes were required to produce a counterfactual on average. Note that multiple changes to the same input token are counted, and that the values are given as percentage of the number of input tokens, i.e. a counterfactual with 25 changes for an input with 20 tokens would result in a number of 125%. This figure shows that kEES has on average the lowest number of changes, which is expected as k is set to two. Other search algorithms produce

counterfactuals with more changes, partly because they can due to they not having such a limit. Of the different search configurations, the line Tokenizer produces the highest number of changes. This is because it produces a lower number of tokens than the Clang Tokenizer, and therefore less changes have to be made and can be made until a counterfactual is found.

The exact search configuration has great influence on the search space. For the kEES, the search space is determined by the cut-off parameter k, but also by the size of the input, which together span up an exponential search tree. The size of the search space of the LIG search is equivalent to the number of tokens in the input. At most, the search iterates over each token, but it aborts as soon as it finds a counterfactual. Other search algorithms are not as dependent on the input. The search space of the GS is determined only by its parameters such as the maximum number of generations, gene pool size, kill ratio and the exact mechanism of the crossover operation. The search space for the greedy search is independent of the input size. Instead, the number of search iterations is determined through the maximum search depth parameter. The exact values for those parameters were determined empirically to give a compromise between the maximal search duration, to allow for a large number of search runs, and a high chance to find counterfactuals within a search run. They are given in Table 7.2.

- **RQ2**: How can gradient information from the model of code influence the search space?

  Figure 4.10 shows that the gradient informed LIG search produces counterfactuals for 10% of search runs. Compared with the greedy search, which is the search algorithm delivering the least number of counterfactuals per search run apart from the LIG search, that yields on average 3.43 counterfactuals, it can be shown that the LIG search does not give an improvement in this regard. The kEES baseline produces on average 285.6 counterfactuals per search run, which is the highest number of the investigated search algorithms.

  The LIG search is also the slowest search algorithm. It takes on average 97.21s to find a counterfactual (2.55s for the GS, 4.46s for the greedy search, 6.94s for kEES). However, in this comparison the LIG search suffers from the low number of search runs for which it produces counterfactuals, as these measurements also include the run times in which it did not produce any.

  When the gradient informed search is compared with the other search algorithms for how long it took to deliver the first counterfactual, it outperforms kEES for every search configuration and the GS when the `recompute_attributions_for_each_iteration` is set to `true`. Depending on the exact search parameters, the greedy search is either significantly faster (0.61s for greedy search with mutation Perturber vs. 1.55s and 2.13s for LIG) or slower (4.46s for greedy search with masked Perturber). The exact values are displayed in Figure 4.9.

- **RQ3**: How do different perturbations functions influence the search space?

  As discussed in Section 3.2.1, the perturbation functions for this thesis are a masked Perturber, where tokens are replaced with the output of a MLM, a mutation Perturber where tokens are replaced with a random token from the dictionary, and a remove token Perturber, which removes tokens.

  These are paired with either a line Tokenizer or a Clang Tokenizer, according to Section 3.2.1. The former one splits the inputs by newline characters, the latter one splits the input by C++ source code tokens according to Clang.

  Since the line Tokenizer produces less tokens than the Clang Tokenizer, it spans up a smaller search space. This smaller search space does not translate into a faster search per counterfactual for all search algorithms, as displayed in Figure 4.4. Instead, the GS is faster when using the Clang Tokenizer (4.48s for line Tokenizer vs. 3.77s for the Clang Tokenizer). This can be explained by the search algorithm being able to make smaller, but more directed changes to the input. kEES and the greedy search do have a lower search duration per counterfactual with the line Tokenizer (11.9s and 2.06s vs 586.39s and 6.66s, respectively). The significant increase of the search duration for kEES stems from the exponential nature of this algorithm, whereas the maximum search duration of the GS and greedy search are determined through hyperparameters, and not only through the problem size.

  Due to the increased computational demand introduced through the MLM, the masked Perturber is slower for any search algorithm than the mutation Perturber and remove token Perturber. This effect is greatest for the kEES algorithm, where it takes on average 819.58s to find a counterfactual with the masked Perturber, 73.24s with the remove token Perturber, and 34.05s with the mutation Perturber. This trend also holds true for both the greedy search and the GS (9.55s vs. 2.1s for remove token Perturber and 1.85s for mutation Perturber, and 7.02s vs. 3.07s and 3.13s, respectively). The difference between the remove token Perturber and mutation Perturber is smaller, however the mutation Perturber is faster for any search algorithm. Even though the removal of tokens might decrease the size of the search space, it can also make any counterfactual candidate to appear more vulnerable to a model of code and decrease its similarity to proper C++ source code. This in turn leads to an increased number of candidates that do not lead to a counterfactual, and therefore longer search durations and a smaller number of counterfactuals.

  The different search configurations have limited influence on the average string similarity. No search configuration is significantly better than any other. kEES achieves the highest similarity, but is not consistently able to outperform the GS. The greedy search scores a higher similarity than the LIG search, but is still consistently worse than the other search algorithms.

6.2 Contributions

The main contributions of this thesis are the following:

- An open source framework to search for and report counterfactuals, available on GitHub[1]. This framework is designed to be easily expandable with new search algorithms, Perturbers, Tokenizers, and models of code.
- An evaluation of currently available open source models of code and search algorithms with respect to their ability to find counterfactuals for C++ source code.
- A discussion on the results and limiting factors of existing technologies, as well as possible avenues for further research in the area of search strategies for counterfactuals.

6.3 Limitations and Future Work

As of writing this thesis, the best model of code available achieves an accuracy of 68.78%. This model is the Microsoft CodeReviewer, as described in Section 4.1.1. Since the authors of CodeReviewer have not published the fine tuned checkpoints achieving this accuracy, an attempt at manually finetuning the model with the provided training data has been made. However, this attempt was unsuccessful at reproducing said accuracy, and it did not deliver useful results on source code other than the training or testing data. The next best model is CodeT5 as discussed in Section 4.1.4, which achieved an accuracy of 65.78%. Other ML models, such as [35] which claims an accuracy of 90% and a processing time of 20ms per file, can only find a very limited subset of potential errors. Namely, [35] is specialised at finding name based bugs in JavaScript source code, and is therefore not comparable to the models used in this thesis which are aimed at finding a greater variety of bugs and vulnerabilities in C++ code.

Both heuristic search algorithms implemented in this thesis, GA and the Greedy Search, rely on accurate and stable heuristics to guide them in their search. None of the models are able to consistently satisfy these requirements, especially on out of distribution data, which leads to a degraded search performance, increasing the time to find counterfactuals, and leading to both false positive and false negative results. The models used in this thesis perform worse than the accuracies claimed in their respective papers. This is evident from the data in Table 4.2, where the models incorrectly classified between 43.18% and 47.37% of source code samples from the validation set. Furthermore, no search configuration using VulBERTA-MLP found a single counterfactual, which indicates that the model classified all perturbations as vulnerable. Table 4.2 shows that the model in principle can classify source code as invulnerable, as 47.37% of the source code samples

[1] https://github.com/chrfwow/MSC

from the CodeXGLUE dataset labeled as vulnerable were not classified as such. This result could indicate that this model overfits on its training data.

Due to the exponential nature of the kEES algorithm, the parameter k had to be fixed to a comparatively low value of 2. Higher values slowed the search down prohibitively, and would have reduced the number of code samples that could have been evaluated too much. This restriction of k limits the number of counterfactuals produced. It also means that only counterfactuals with a low number of changes are found, which are closer to the original input and therefore preferable.

As already discussed, the models of code available at the time or writing this thesis are not stable enough to properly guide heuristic search strategies. However, the fast advance of transformer based models of code and other architectures in recent times makes it plausible that their quality will improve in the near future, and lead to models that are capable of providing sufficient guidance for heuristic approaches. Furthermore, it can be assumed that future models will be able to classify input quicker, which will speed up the search for counterfactuals, and allow for a more thorough exploration of the search space. Currently, such a thorough exploration is not feasible, as demonstrated with the poor temporal performance of the kEES algorithm as described in Section 4.4.

A different, if more complex, post processing step could be added to the Perturbers, as opposed to the current code formatting. This new post processing step could attempt to fix syntax errors introduced through the perturbation function. With syntax errors removed, the models of code should classify more counterfactual candidates as invulnerable and therefore deliver more and more relevant counterfactuals.

As discussed in Section 1, the usefulness of counterfactuals depends on its closeness to the input. This calculation is done in this thesis. However, an even more expressive evaluation of the usefulness of counterfactuals would be to ask software developers for their opinion of counterfactuals, as was done by Cito et al. [8]. Unfortunately, due to the large number of different search configuration and the lack of available software developers to ask, no such qualitative assessment could be done. In the future, this evaluation could be done if more resources are available.

6.4 Threads to Validity

This section discusses both external and internal threads to the validity of this thesis.

6.4.1 Threads to External Validity

External validity describes how well the findings of this thesis can be applied outside of its context. This thesis only considers the search for counterfactuals for vulnerable source code snippets from the CodeXGLUE dataset. The CodeXGLUE dataset only contains source code in the C++ programing language, but from two large open source projects, FFMPEG and Qemu, which should provide enough diversity in the data to also represent

other C++ projects. Additionally, within this investigation, a large array of different models of code, perturbation functions, tokenizers, and search algorithms are used. This gives confidence in the thoroughness of the performed investigation. In principle, an efficient search for counterfactuals could be needed in any domain where black box tools are used. For these unrelated cases, the resulting frameworks presents a starting point for further studies. In the domain of detection of vulnerable source code, however, the used models are the best open source models available at the time of writing of this thesis, and the search algorithms are descriptive examples. Therefore it can be assumed that the results do hold true also when applied to other programming languages, new models of code, or new search algorithms.

6.4.2 Threads to Internal Validity

Internal validity is when the findings of this thesis are actually explained through the data and explanations provided, and not through some other means.

The greatest cause of concern is the poor performance of the VulBERTa-MLP model of code, which could not find any counterfactuals. This could indicate some bug in the usage of the model. However, all other models did lead to counterfactuals, which indicates that the search itself works as expected. Furthermore, VulBERTa did correctly classify 50 of the source code snippets from the CodeXGLUE dataset labelled as vulnerable, and incorrectly classified 45 additional snippets. This shows that the way the model is used in this thesis, it is in principle able to classify source code snippets as both vulnerable and invulnerable. Therefore it can be assumed, that the model does not perform well on source code not present in the CodeXGLUE dataset, which would result from the perturbations generated during the counterfactual search. Such a behaviour is one that is sometimes encountered in machine learning and is called overfitting. If the model indeed overfits, this could be a sign of it being trained for too long or on too little data, neither of which could be changed within the scope of this thesis.

Another cause of concern is the poor accuracy of all models of code used in this thesis. As an example, CodeT5 should achieve an accuracy of 65.78% on the CodeXGLUE benchmark according to [47]. In this thesis, it incorrectly classified 44% of all source code samples, which translates to an accuracy of 56%, much worse than the claimed one. However, for the accuracy analysis in [47], both vulnerable and invulnerable source code samples from the CodeXGLUE datasets were taken into account. This is not true for this thesis, where only source code snippets labelled as vulnerable were used. If CodeT5 is much better at correctly identifying invulnerable snippets, such results could occur. Additionally, the CodeXGLUE dataset contains 12460 snippets labelled as vulnerable, and 14858 labelled as invulnerable. If the model performed better on invulnerable snippets, naturally a greater accuracy would be reported, just because more of those samples exist in the dataset. These findings can also be applied to the other models used in this thesis.

CHAPTER 7

Appendix

7.1 Search Configurations

Abbreviation	Classifier	Search Algorithm	Perturber	Tokenizer	Unmasker
CB GA M C CB	CodeBert	Genetic	Masked	Clang	CodeBert
CB GA M L CB	CodeBert	Genetic	Masked	Line	CodeBert
CB GA M C NO	CodeBert	Genetic	Mutation	Clang	NoOp
CB GA M L NO	CodeBert	Genetic	Mutation	Line	NoOp
CB GA RT C NO	CodeBert	Genetic	RemoveToken	Clang	NoOp
CB GA RT L NO	CodeBert	Genetic	RemoveToken	Line	NoOp
CB Gr M C CB	CodeBert	Greedy	Masked	Clang	CodeBert
CB Gr M L CB	CodeBert	Greedy	Masked	Line	CodeBert
CB Gr M C NO	CodeBert	Greedy	Mutation	Clang	NoOp
CB Gr M L NO	CodeBert	Greedy	Mutation	Line	NoOp
CB Gr RT C NO	CodeBert	Greedy	RemoveToken	Clang	NoOp
CB Gr RT L NO	CodeBert	Greedy	RemoveToken	Line	NoOp
CB KEES M C CB	CodeBert	*k*EES	Masked	Clang	CodeBert
CB KEES M L CB	CodeBert	*k*EES	Masked	Line	CodeBert
CB KEES M C NO	CodeBert	*k*EES	Mutation	Clang	NoOp
CB KEES M L NO	CodeBert	*k*EES	Mutation	Line	NoOp
CB KEES RT C NO	CodeBert	*k*EES	RemoveToken	Clang	NoOp
CB KEES RT L NO	CodeBert	*k*EES	RemoveToken	Line	NoOp
CB LS	CodeBert	LigSearch	n.a.	n.a.	n.a.
CB LS	CodeBert	LigSearch	n.a.	n.a.	n.a.
CT5 GA M C CB	CodeT5	Genetic	Masked	Clang	CodeBert
CT5 GA M L CB	CodeT5	Genetic	Masked	Line	CodeBert
CT5 GA M C NO	CodeT5	Genetic	Mutation	Clang	NoOp

CT5 GA M L NO	CodeT5	Genetic	Mutation	Line	NoOp
CT5 GA RT C NO	CodeT5	Genetic	RemoveToken	Clang	NoOp
CT5 GA RT L NO	CodeT5	Genetic	RemoveToken	Line	NoOp
CT5 Gr M C CB	CodeT5	Greedy	Masked	Clang	CodeBert
CT5 Gr M L CB	CodeT5	Greedy	Masked	Line	CodeBert
CT5 Gr M C NO	CodeT5	Greedy	Mutation	Clang	NoOp
CT5 Gr M L NO	CodeT5	Greedy	Mutation	Line	NoOp
CT5 Gr RT C NO	CodeT5	Greedy	RemoveToken	Clang	NoOp
CT5 Gr RT L NO	CodeT5	Greedy	RemoveToken	Line	NoOp
CT5 KEES M C CB	CodeT5	*k*EES	Masked	Clang	CodeBert
CT5 KEES M L CB	CodeT5	*k*EES	Masked	Line	CodeBert
CT5 KEES M C NO	CodeT5	*k*EES	Mutation	Clang	NoOp
CT5 KEES M L NO	CodeT5	*k*EES	Mutation	Line	NoOp
CT5 KEES RT C NO	CodeT5	*k*EES	RemoveToken	Clang	NoOp
CT5 KEES RT L NO	CodeT5	*k*EES	RemoveToken	Line	NoOp
PLB GA M C CB	PLBart	Genetic	Masked	Clang	CodeBert
PLB GA M L CB	PLBart	Genetic	Masked	Line	CodeBert
PLB GA M C NO	PLBart	Genetic	Mutation	Clang	NoOp
PLB GA M L NO	PLBart	Genetic	Mutation	Line	NoOp
PLB GA RT C NO	PLBart	Genetic	RemoveToken	Clang	NoOp
PLB GA RT L NO	PLBart	Genetic	RemoveToken	Line	NoOp
PLB Gr M C CB	PLBart	Greedy	Masked	Clang	CodeBert
PLB Gr M L CB	PLBart	Greedy	Masked	Line	CodeBert
PLB Gr M C NO	PLBart	Greedy	Mutation	Clang	NoOp
PLB Gr M L NO	PLBart	Greedy	Mutation	Line	NoOp
PLB Gr RT C NO	PLBart	Greedy	RemoveToken	Clang	NoOp
PLB Gr RT L NO	PLBart	Greedy	RemoveToken	Line	NoOp
PLB KEES M C CB	PLBart	*k*EES	Masked	Clang	CodeBert
PLB KEES M L CB	PLBart	*k*EES	Masked	Line	CodeBert
PLB KEES M C NO	PLBart	*k*EES	Mutation	Clang	NoOp
PLB KEES M L NO	PLBart	*k*EES	Mutation	Line	NoOp
PLB KEES RT C NO	PLBart	*k*EES	RemoveToken	Clang	NoOp
PLB KEES RT L NO	PLBart	*k*EES	RemoveToken	Line	NoOp
PLB LS	PLBart	LigSearch	n.a.	n.a.	n.a.
PLB LS	PLBart	LigSearch	n.a.	n.a.	n.a.
VB GA M C CB	VulBERTa	Genetic	Masked	Clang	CodeBert
VB GA M L CB	VulBERTa	Genetic	Masked	Line	CodeBert
VB GA M C NO	VulBERTa	Genetic	Mutation	Clang	NoOp
VB GA M L NO	VulBERTa	Genetic	Mutation	Line	NoOp
VB GA RT C NO	VulBERTa	Genetic	RemoveToken	Clang	NoOp
VB GA RT L NO	VulBERTa	Genetic	RemoveToken	Line	NoOp

VB Gr M C CB	VulBERTa	Greedy	Masked	Clang	CodeBert
VB Gr M L CB	VulBERTa	Greedy	Masked	Line	CodeBert
VB Gr M C NO	VulBERTa	Greedy	Mutation	Clang	NoOp
VB Gr M L NO	VulBERTa	Greedy	Mutation	Line	NoOp
VB Gr RT C NO	VulBERTa	Greedy	RemoveToken	Clang	NoOp
VB Gr RT L NO	VulBERTa	Greedy	RemoveToken	Line	NoOp
VB KEES M C CB	VulBERTa	*k*EES	Masked	Clang	CodeBert
VB KEES M L CB	VulBERTa	*k*EES	Masked	Line	CodeBert
VB KEES M C NO	VulBERTa	*k*EES	Mutation	Clang	NoOp
VB KEES M L NO	VulBERTa	*k*EES	Mutation	Line	NoOp
VB KEES RT C NO	VulBERTa	*k*EES	RemoveToken	Clang	NoOp
VB KEES RT L NO	VulBERTa	*k*EES	RemoveToken	Line	NoOp
VB LS	VulBERTa	LigSearch	n.a.	n.a.	n.a.
VB LS	VulBERTa	LigSearch	n.a.	n.a.	n.a.

Table 7.1: The abbreviations for all combinations of search configurations.

7.2 Search Parameters

Abbreviation	Iterations	Gene pool size	Kill ratio	Max age	Max survivors	Steps per iteration	Recompute attributions	Max tokens removal ratio
CB GA M C CB	10	40	0.3	n.a.	n.a.	n.a.	n.a.	n.a.
CB GA M L CB	10	40	0.3	n.a.	n.a.	n.a.	n.a.	n.a.
CB GA M C NO	10	40	0.3	n.a.	n.a.	n.a.	n.a.	n.a.
CB GA M L NO	10	40	0.3	n.a.	n.a.	n.a.	n.a.	n.a.
CB GA RT C NO	10	40	0.3	n.a.	n.a.	n.a.	n.a.	n.a.
CB GA RT L NO	10	40	0.3	n.a.	n.a.	n.a.	n.a.	n.a.
CB Gr M C CB	30	n.a.	n.a.	25	10	n.a.	n.a.	n.a.
CB Gr M L CB	30	n.a.	n.a.	25	10	n.a.	n.a.	n.a.
CB Gr M C NO	30	n.a.	n.a.	25	10	n.a.	n.a.	n.a.
CB Gr M L NO	30	n.a.	n.a.	25	10	n.a.	n.a.	n.a.
CB Gr RT C NO	30	n.a.	n.a.	25	10	n.a.	n.a.	n.a.
CB Gr RT L NO	30	n.a.	n.a.	25	10	n.a.	n.a.	n.a.

CB KEES M C CB	2	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
CB KEES M L CB	2	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
CB KEES M C NO	2	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
CB KEES M L NO	2	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
CB KEES RT C NO	2	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
CB KEES RT L NO	2	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
CB LS	100	n.a.	n.a.	n.a.	n.a.	20	True	0.6
CB LS	100	n.a.	n.a.	n.a.	n.a.	20	False	0.6
CT5 GA M C CB	10	40	0.3	n.a.	n.a.	n.a.	n.a.	n.a.
CT5 GA M L CB	10	40	0.3	n.a.	n.a.	n.a.	n.a.	n.a.
CT5 GA M C NO	10	40	0.3	n.a.	n.a.	n.a.	n.a.	n.a.
CT5 GA M L NO	10	40	0.3	n.a.	n.a.	n.a.	n.a.	n.a.
CT5 GA RT C NO	10	40	0.3	n.a.	n.a.	n.a.	n.a.	n.a.
CT5 GA RT L NO	10	40	0.3	n.a.	n.a.	n.a.	n.a.	n.a.
CT5 Gr M C CB	30	n.a.	n.a.	25	10	n.a.	n.a.	n.a.

CT5 Gr M L CB	30	n.a.	n.a.	25	10	n.a.	n.a.	n.a.
CT5 Gr M C NO	30	n.a.	n.a.	25	10	n.a.	n.a.	n.a.
CT5 Gr M L NO	30	n.a.	n.a.	25	10	n.a.	n.a.	n.a.
CT5 Gr RT C NO	30	n.a.	n.a.	25	10	n.a.	n.a.	n.a.
CT5 Gr RT L NO	30	n.a.	n.a.	25	10	n.a.	n.a.	n.a.
CT5 KEES M C CB	2	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
CT5 KEES M L CB	2	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
CT5 KEES M C NO	2	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
CT5 KEES M L NO	2	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
CT5 KEES RT C NO	2	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
CT5 KEES RT L NO	2	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
PLB GA M C CB	10	40	0.3	n.a.	n.a.	n.a.	n.a.	n.a.
PLB GA M L CB	10	40	0.3	n.a.	n.a.	n.a.	n.a.	n.a.
PLB GA M C NO	10	40	0.3	n.a.	n.a.	n.a.	n.a.	n.a.

PLB GA M L NO	10	40	0.3	n.a.	n.a.	n.a.	n.a.	n.a.
PLB GA RT C NO	10	40	0.3	n.a.	n.a.	n.a.	n.a.	n.a.
PLB GA RT L NO	10	40	0.3	n.a.	n.a.	n.a.	n.a.	n.a.
PLB Gr M C CB	30	n.a.	n.a.	25	10	n.a.	n.a.	n.a.
PLB Gr M L CB	30	n.a.	n.a.	25	10	n.a.	n.a.	n.a.
PLB Gr M C NO	30	n.a.	n.a.	25	10	n.a.	n.a.	n.a.
PLB Gr M L NO	30	n.a.	n.a.	25	10	n.a.	n.a.	n.a.
PLB Gr RT C NO	30	n.a.	n.a.	25	10	n.a.	n.a.	n.a.
PLB Gr RT L NO	30	n.a.	n.a.	25	10	n.a.	n.a.	n.a.
PLB KEES M C CB	2	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
PLB KEES M L CB	2	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
PLB KEES M C NO	2	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
PLB KEES M L NO	2	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
PLB KEES RT C NO	2	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.

PLB KEES RT L NO	2	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
PLB LS	100	n.a.	n.a.	n.a.	n.a.	20	True	0.6
PLB LS	100	n.a.	n.a.	n.a.	n.a.	20	False	0.6
VB GA M C CB	10	40	0.3	n.a.	n.a.	n.a.	n.a.	n.a.
VB GA M L CB	10	40	0.3	n.a.	n.a.	n.a.	n.a.	n.a.
VB GA M C NO	10	40	0.3	n.a.	n.a.	n.a.	n.a.	n.a.
VB GA M L NO	10	40	0.3	n.a.	n.a.	n.a.	n.a.	n.a.
VB GA RT C NO	10	40	0.3	n.a.	n.a.	n.a.	n.a.	n.a.
VB GA RT L NO	10	40	0.3	n.a.	n.a.	n.a.	n.a.	n.a.
VB Gr M C CB	30	n.a.	n.a.	25	10	n.a.	n.a.	n.a.
VB Gr M L CB	30	n.a.	n.a.	25	10	n.a.	n.a.	n.a.
VB Gr M C NO	30	n.a.	n.a.	25	10	n.a.	n.a.	n.a.
VB Gr M L NO	30	n.a.	n.a.	25	10	n.a.	n.a.	n.a.
VB Gr RT C NO	30	n.a.	n.a.	25	10	n.a.	n.a.	n.a.
VB Gr RT L NO	30	n.a.	n.a.	25	10	n.a.	n.a.	n.a.

VB KEES M C CB	2	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
VB KEES M L CB	2	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
VB KEES M C NO	2	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
VB KEES M L NO	2	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
VB KEES RT C NO	2	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
VB KEES RT L NO	2	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
VB LS	100	n.a.	n.a.	n.a.	n.a.	20	True	0.6
VB LS	100	n.a.	n.a.	n.a.	n.a.	20	False	0.6

Table 7.2: Search parameters for every combination of search configuration. Not every parameter is applicable to every configuration. Inapplicable parameters are denoted as "n.a.".

7.3 Search Results

Configuration	Avg. processing time per token [ms]	Avg. number of CFs per token	Total number of CFs	Number of CFs with ≤ 2 changes	Number of CFs with $\leq$ 20% of input tokens changed	Avg. string similarity of input to CF
CB Gr M L CB	392.02	0.24	11	6	6	0.0761
CB GA M L CB	1404.77	1.44	65	11	27	0.167
CB KEES M L CB	2632.36	1.29	58	58	56	0.1042
CB Gr M L NO	67.72	0.13	6	3	2	0.0848
CB GA M L NO	320.77	0.87	39	14	17	0.1678
CB KEES M L NO	299.45	1.38	62	62	62	0.1435
CB Gr RT L NO	69.87	0.22	10	6	5	0.0525
CB GA RT L NO	321.17	0.98	44	13	28	0.1938
CB KEES RT L NO	338.45	1.38	62	62	61	0.1389
CB Gr M C CB	192.12	0.24	11	11	11	0.0907
CB GA M C CB	236.61	0.29	13	7	13	0.0541

CB KEES M C CB	24656.15	75.53	3399	3399	3399	0.2504
CB Gr M C NO	30.19	2.51	113	113	113	0.1251
CB GA M C NO	53.19	2.76	124	86	124	0.1434
CB KEES M C NO	1875.06	103.98	4679	4679	4679	0.216
CB Gr RT C NO	29.57	2.16	97	97	97	0.1084
CB GA RT C NO	53.69	2.04	92	71	86	0.1823
CB KEES RT C NO	1847.93	70.07	3153	3153	3153	0.1472
PLB Gr M L CB	481.16	0.58	26	13	12	0.1525
PLB GA M L CB	1727.95	2.2	99	50	52	0.1621
PLB KEES M L CB	3079.92	1.89	85	85	73	0.2175
PLB Gr M L NO	87.15	0.44	20	17	14	0.1099
PLB GA M L NO	417.62	2.73	123	64	64	0.2144
PLB KEES M L NO	302.99	1.6	72	72	66	0.1948
PLB Gr RT L NO	83.85	0.69	31	13	10	0.1042

PLB GA RT L NO	409.82	1.2	54	20	16	0.1691
PLB KEES RT L NO	345.41	0.91	41	41	32	0.1608
PLB Gr M C CB	206.87	0.8	36	35	36	0.1281
PLB GA M C CB	308.51	2.82	127	67	123	0.1609
PLB KEES M C CB	26528.56	378.78	17045	17045	17045	0.326
PLB Gr M C NO	34.64	3.11	140	139	140	0.1836
PLB GA M C NO	71.85	4.51	203	108	182	0.2204
PLB KEES M C NO	1701.13	534.0	24030	24030	24030	0.2736
PLB Gr RT C NO	34.71	3.58	161	160	161	0.164
PLB GA RT C NO	71.97	4.58	206	89	189	0.2543
PLB KEES RT C NO	1672.99	479.22	21565	21565	21565	0.2556
CB LS	6279.71	0.0	0	0	0	0.0
CB LS	1009.0	0.11	5	0	0	0.0771
PLB LS	2739.27	0.11	5	1	0	0.0916
PLB LS	851.61	0.16	7	2	0	0.1133
CT5 Gr M L CB	294.18	3.84	192	165	169	0.2705

CT5 GA M L CB	1497.91	19.64	982	340	521	0.2555
CT5 KEES M L CB	2116.0	28.68	1434	1434	1431	0.3246
CT5 Gr M L NO	89.62	1.78	89	76	77	0.1991
CT5 GA M L NO	412.99	5.7	285	206	230	0.3171
CT5 KEES M L NO	498.01	13.86	693	693	689	0.3307
CT5 Gr RT L NO	90.56	2.76	138	122	120	0.2438
CT5 GA RT L NO	417.91	4.3	215	114	159	0.3057
CT5 KEES RT L NO	517.67	12.38	619	619	613	0.3315
CT5 Gr M C CB	162.21	8.4	420	420	420	0.2524
CT5 GA M C CB	295.47	22.66	1133	483	1121	0.2343
CT5 KEES M C CB	17992.45	528.42	26421	26421	26421	0.3678
CT5 Gr M C NO	43.07	7.94	397	397	397	0.2633
CT5 GA M C NO	80.25	13.44	672	402	670	0.3434
CT5 KEES M C NO	3238.37	1018.68	50934	50934	50934	0.3435

CT5 Gr RT C NO	44.38	8.64	432	432	432	0.3103
CT5 GA RT C NO	72.24	11.28	564	336	564	0.3256
CT5 KEES RT C NO	3077.21	797.06	39853	39853	39853	0.3452
VB Gr M L CB	535.46	0.0	0	0	0	0.0
VB GA M L CB	1726.25	0.0	0	0	0	0.0
VB KEES M L CB	6522.95	0.0	0	0	0	0.0
VB Gr M L NO	114.77	0.0	0	0	0	0.0
VB GA M L NO	521.37	0.0	0	0	0	0.0
VB KEES M L NO	741.55	0.0	0	0	0	0.0
VB Gr RT L NO	117.39	0.0	0	0	0	0.0
VB GA RT L NO	518.05	0.0	0	0	0	0.0
VB KEES RT L NO	781.36	0.0	0	0	0	0.0
VB Gr M C CB	301.65	0.0	0	0	0	0.0
VB GA M C CB	327.8	0.0	0	0	0	0.0

VB KEES M C CB	46813.41	0.0	0	0	0	0.0
VB Gr M C NO	55.41	0.0	0	0	0	0.0
VB GA M C NO	92.76	0.0	0	0	0	0.0
VB KEES M C NO	5567.43	0.0	0	0	0	0.0
VB Gr RT C NO	56.71	0.0	0	0	0	0.0
VB GA RT C NO	93.15	0.0	0	0	0	0.0
VB KEES RT C NO	5099.97	0.0	0	0	0	0.0
VB LS	108393.41	0.0	0	0	0	0.0
VB LS	1064.97	0.0	0	0	0	0.0

Table 7.3: The results from every combination of search algorithm, tokenizer, perturber, unmasker and classifier. String similarity is calculated using the Jaro distance [4], where 1 means identical strings, and 0 means no similarity between two strings.

7.4 Removed Tokens Categories

Config.	FUNC_DEF [%]	KEYWORD [%]	PUNCT. [%]	UNKNOWN [%]	LOOP [%]	IDENT. [%]	LITERAL [%]
CB Gr M L CB	44.44	44.44	11.11	-	-	-	-
CB GA M L CB	55.56	27.78	16.67	-	-	-	-
CB KEES M L CB	45.61	21.93	18.42	14.04	-	-	-
CB Gr M L NO	40.0	20.0	40.0	-	-	-	-
CB GA M L NO	45.83	25.0	25.0	4.17	-	-	-
CB KEES M L NO	34.15	24.39	35.77	5.69	-	-	-
CB Gr RT L NO	50.0	30.0	20.0	-	-	-	-
CB GA RT L NO	37.5	33.33	29.17	-	-	-	-
CB KEES RT L NO	48.36	18.85	23.77	6.56	2.46	-	-
CB Gr M C CB	-	18.18	36.36	-	-	45.45	-
CB GA M C CB	-	-	81.82	-	-	18.18	-
CB KEES M C CB	-	4.31	58.24	3.85	0.15	27.64	5.82
CB Gr M C NO	-	13.27	30.97	5.31	-	46.9	3.54

CB GA M C NO	-	5.15	59.79	-	-	32.99	2.06
CB KEES M C NO	-	3.53	55.45	0.36	0.05	36.23	4.38
CB Gr RT C NO	-	8.25	26.8	6.19	-	52.58	6.19
CB GA RT C NO	-	8.11	55.41	-	-	25.68	10.81
CB KEES RT C NO	-	2.85	63.06	0.3	-	29.9	3.89
PLB Gr M L CB	70.59	11.76	11.76	5.88	-	-	-
PLB GA M L CB	54.79	24.66	10.96	2.74	6.85	-	-
PLB KEES M L CB	42.86	25.47	25.47	3.73	2.48	-	-
PLB Gr M L NO	50.0	31.82	18.18	-	-	-	-
PLB GA M L NO	52.53	16.16	19.19	9.09	3.03	-	-
PLB KEES M L NO	44.78	21.64	18.66	5.22	9.7	-	-
PLB Gr RT L NO	55.0	30.0	15.0	-	-	-	-
PLB GA RT L NO	70.59	23.53	5.88	-	-	-	-
PLB KEES RT L NO	55.84	27.27	10.39	3.9	2.6	-	-

PLB Gr M C CB	-	2.86	31.43	2.86	5.71	51.43	5.71
PLB GA M C CB	-	0.93	59.26	-	-	31.48	8.33
PLB KEES M C CB	-	2.39	61.82	0.6	0.97	30.82	3.41
PLB Gr M C NO	-	7.91	25.18	7.19	0.72	48.92	10.07
PLB GA M C NO	-	5.71	55.71	-	0.71	31.43	6.43
PLB KEES M C NO	-	3.39	52.04	1.75	0.19	37.19	5.43
PLB Gr RT C NO	-	10.0	21.88	8.12	0.62	50.0	9.38
PLB GA RT C NO	-	3.15	55.91	-	-	25.98	14.96
PLB KEES RT C NO	-	2.62	55.97	1.96	0.3	35.97	3.18
CB LS	-	-	-	-	-	-	-
CB LS	-	-	-	-	-	-	-
PLB LS	-	-	-	-	-	100.0	-
PLB LS	-	-	-	-	-	100.0	-
CT5 Gr M L CB	28.16	25.73	31.07	12.62	1.94	0.49	-
CT5 GA M L CB	35.79	26.18	30.27	6.54	1.23	-	-
CT5 KEES M L CB	31.96	26.85	28.5	9.45	3.24	-	-

CT5 Gr M L NO	32.5	21.25	30.0	13.75	2.5	-	-
CT5 GA M L NO	30.95	29.37	21.43	17.06	1.19	-	-
CT5 KEES M L NO	21.48	39.97	22.3	13.2	2.98	0.07	-
CT5 Gr RT L NO	31.11	35.56	19.26	10.37	3.7	-	-
CT5 GA RT L NO	35.53	26.97	23.03	9.87	4.61	-	-
CT5 KEES RT L NO	24.19	37.69	22.56	11.88	3.68	-	-
CT5 Gr M C CB	-	19.29	30.48	2.62	0.48	40.24	6.9
CT5 GA M C CB	-	10.26	53.9	0.14	-	33.65	2.05
CT5 KEES M C CB	-	7.62	56.82	0.88	0.35	30.53	3.8
CT5 Gr M C NO	-	14.32	26.63	1.01	1.01	50.5	6.53
CT5 GA M C NO	-	6.94	60.82	-	0.2	27.76	4.29
CT5 KEES M C NO	-	4.25	57.36	0.86	0.94	32.8	3.8
CT5 Gr RT C NO	-	13.89	29.17	0.93	0.93	48.61	6.48
CT5 GA RT C NO	-	7.98	61.1	-	-	28.18	2.74

CT5 KEES RT C NO	-	3.52	58.05	1.31	1.06	32.82	3.24
VB Gr M L CB	-	-	-	-	-	-	-
VB GA M L CB	-	-	-	-	-	-	-
VB KEES M L CB	-	-	-	-	-	-	-
VB Gr M L NO	-	-	-	-	-	-	-
VB GA M L NO	-	-	-	-	-	-	-
VB KEES M L NO	-	-	-	-	-	-	-
VB Gr RT L NO	-	-	-	-	-	-	-
VB GA RT L NO	-	-	-	-	-	-	-
VB KEES RT L NO	-	-	-	-	-	-	-
VB Gr M C CB	-	-	-	-	-	-	-
VB GA M C CB	-	-	-	-	-	-	-
VB KEES M C CB	-	-	-	-	-	-	-
VB Gr M C NO	-	-	-	-	-	-	-

VB GA M C NO	-	-	-	-	-	-	-
VB KEES M C NO	-	-	-	-	-	-	-
VB Gr RT C NO	-	-	-	-	-	-	-
VB GA RT C NO	-	-	-	-	-	-	-
VB KEES RT C NO	-	-	-	-	-	-	-
VB LS	-	-	-	-	-	-	-
VB LS	-	-	-	-	-	-	-

Table 7.4: The percentage of counterfactuals with ≤ 2 changes to the input with removals containing the types of program statements in the table. Categories of removed tokens are explained in Section 4.2.5.

List of Figures

List of Tables

Acronyms

kEES k-Exponential Exhaustive Search. 2, 11, 23, 24, 27–31, 36, 37, 49–51, 53

<EOS> End Of Sequence. 6

<SOS> Start Of Sequence. 6

AI Artificial Intelligence. 1

CNN Convolutional Neural Network. 30

CPU Central Processing Unit. 16, 31

GA Genetic Algorithm. 2, 8–11, 21, 22, 52

GPT Generative Pretrained Transformer. 6–8

GPU Graphics Processing Unit. 16, 31

GS Genetic Search. 8, 9, 11, 21, 22, 27–31, 37, 50, 51

IG Integrated Gradients. 11, 12

LIG Layer Integrated Gradients. 3, 11, 16, 24, 25, 31, 32, 35, 37, 41–44, 49–51, 78

LLM Large Language Model. 6

ML Machine Learning. 1, 8, 15, 52

MLM Masked Language Model. 3, 20, 21, 28, 32, 47, 51

MLP Multi Layer Perceptron. 30

NEAT NeuroEvolution of Augmenting Topologies. 10, 11

NN Neural Network. 11, 12, 31, 45

RLHF Reinforcement Learning from Human Feedback. 7

RNN Recurrent Neural Network. 5

T5 Text-To-Text Transfer Transformer. 30

Bibliography

[1] J. Achiam et al. *GPT-4 Technical Report*. 2023-12-18. DOI: 10.48550/arXiv.2303.08774. arXiv: 2303.08774[cs]. URL: http://arxiv.org/abs/2303.08774 (visited on 2024-02-22).

[2] W. U. Ahmad et al. *Unified Pre-training for Program Understanding and Generation*. 2021-04-10. DOI: 10.48550/arXiv.2103.06333. arXiv: 2103.06333[cs]. URL: http://arxiv.org/abs/2103.06333 (visited on 2024-04-04).

[3] A. Albalak et al. *A Survey on Data Selection for Language Models*. 2024-02-26. DOI: 10.48550/arXiv.2402.16827. arXiv: 2402.16827[cs]. URL: http://arxiv.org/abs/2402.16827 (visited on 2024-02-28).

[4] J. Basak et al. "On Computing the Jaro Similarity Between Two Strings". In: *Bioinformatics Research and Applications*. Ed. by X. Guo et al. Singapore: Springer Nature, 2023, pp. 31–44. ISBN: 978-981-9970-74-2. DOI: 10.1007/978-981-99-7074-2_3.

[5] P. E. Black. *greedy algorithm*. greedy algorithm. 2005-02-02. URL: https://xlinux.nist.gov/dads//HTML/greedyalgo.html (visited on 2024-04-20).

[6] N. Carlini et al. *Stealing Part of a Production Language Model*. 2024-03-11. DOI: 10.48550/arXiv.2403.06634. arXiv: 2403.06634[cs]. URL: http://arxiv.org/abs/2403.06634 (visited on 2024-04-08).

[7] I. Čík et al. "Explaining Deep Neural Network using Layer-wise Relevance Propagation and Integrated Gradients". In: *2021 IEEE 19th World Symposium on Applied Machine Intelligence and Informatics (SAMI)*. 2021 IEEE 19th World Symposium on Applied Machine Intelligence and Informatics (SAMI). 2021-01, pp. 000381–000386. DOI: 10.1109/SAMI50585.2021.9378686. URL: https://ieeexplore.ieee.org/abstract/document/9378686 (visited on 2024-04-27).

[8] J. Cito et al. "Counterfactual explanations for models of code". In: *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP '22. New York, NY, USA: Association for Computing Machinery, 2022-10-17, pp. 125–134. ISBN: 978-1-4503-9226-6. DOI: 10.1145/3510457.3513081. URL: https://dl.acm.org/doi/10.1145/3510457.3513081 (visited on 2023-05-19).

[9] T. H. Cormen et al. *Introduction To Algorithms*. Google-Books-ID: NLngYyWFl_YC. MIT Press, 2001. 1216 pp. ISBN: 978-0-262-03293-3.

[10] K. Deb. "An introduction to genetic algorithms". In: *Sadhana* 24.4 (1999-08-01), pp. 293–315. ISSN: 0973-7677. DOI: 10.1007/BF02823145. URL: https://doi.org/10.1007/BF02823145 (visited on 2024-02-23).

[11] J. Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019-05-24. DOI: 10.48550/arXiv.1810.04805. arXiv: 1810.04805[cs]. URL: http://arxiv.org/abs/1810.04805 (visited on 2024-02-22).

[12] Z. Feng et al. *CodeBERT: A Pre-Trained Model for Programming and Natural Languages*. 2020-09-18. DOI: 10.48550/arXiv.2002.08155. arXiv: 2002.08155[cs]. URL: http://arxiv.org/abs/2002.08155 (visited on 2024-01-26).

[13] S. A. Fezza et al. "Perceptual Evaluation of Adversarial Attacks for CNN-based Image Classification". In: *2019 Eleventh International Conference on Quality of Multimedia Experience (QoMEX)*. 2019 Eleventh International Conference on Quality of Multimedia Experience (QoMEX). ISSN: 2472-7814. 2019-06, pp. 1–6. DOI: 10.1109/QoMEX.2019.8743213. URL: https://ieeexplore.ieee.org/abstract/document/8743213 (visited on 2024-02-28).

[14] L. Floridi and M. Chiriatti. "GPT-3: Its Nature, Scope, Limits, and Consequences". In: *Minds and Machines* 30.4 (2020-12-01), pp. 681–694. ISSN: 1572-8641. DOI: 10.1007/s11023-020-09548-1. URL: https://doi.org/10.1007/s11023-020-09548-1 (visited on 2024-02-23).

[15] A. A. Freitas. "Comprehensible classification models: a position paper". In: *ACM SIGKDD Explorations Newsletter*. Vol. 15. 2014-03-17, pp. 1–10. DOI: 10.1145/2594473.2594475. URL: https://dl.acm.org/doi/10.1145/2594473.2594475 (visited on 2023-05-19).

[16] D. Gragnaniello et al. "Analysis of Adversarial Attacks against CNN-based Image Forgery Detectors". In: *2018 26th European Signal Processing Conference (EUSIPCO)*. 2018 26th European Signal Processing Conference (EUSIPCO). ISSN: 2076-1465. 2018-09, pp. 967–971. DOI: 10.23919/EUSIPCO.2018.8553560. URL: https://ieeexplore.ieee.org/abstract/document/8553560 (visited on 2024-02-28).

[17] R. Guidotti. "Counterfactual explanations and how to find them: literature review and benchmarking". In: *Data Mining and Knowledge Discovery*. 2022-04-28. DOI: 10.1007/s10618-022-00831-6. URL: https://doi.org/10.1007/s10618-022-00831-6 (visited on 2023-05-19).

[18] R. Guidotti et al. "A Survey of Methods for Explaining Black Box Models". In: *ACM Computing Surveys* 51.5 (2018-08-22), 93:1–93:42. ISSN: 0360-0300. DOI: 10.1145/3236009. URL: https://dl.acm.org/doi/10.1145/3236009 (visited on 2024-02-29).

[19] J. Han et al. “An Empirical Study of the Dependency Networks of Deep Learning Libraries”. In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME). ISSN: 2576-3148. 2020-09, pp. 868–878. DOI: 10.1109/ICSME46990.2020.00116. URL: https://ieeexplore.ieee.org/document/9240645 (visited on 2024-04-26).

[20] H. Hanif and S. Maffeis. “VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection”. In: *2022 International Joint Conference on Neural Networks (IJCNN)*. 2022 International Joint Conference on Neural Networks (IJCNN). ISSN: 2161-4407. 2022-07, pp. 1–8. DOI: 10.1109/IJCNN55064.2022.9892280. URL: https://ieeexplore.ieee.org/document/9892280 (visited on 2024-04-07).

[21] E. Horton and C. Parnin. “Gistable: Evaluating the Executability of Python Code Snippets on GitHub”. In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). ISSN: 2576-3148. 2018-09, pp. 217–227. DOI: 10.1109/ICSME.2018.00031. URL: https://ieeexplore.ieee.org/abstract/document/8530031?casa_token=Kqeg7i9xtusAAAAA:_HoLfwRlfHfmB7rHP2OYMqztHDeEcVbgXmNW5t3_qA593f2Dr0ifpSKCCGkt_4ClqPcGZ7tB (visited on 2024-04-26).

[22] M. A. Jaro. “Advances in Record-Linkage Methodology as Applied to Matching the 1985 Census of Tampa, Florida”. In: *Journal of the American Statistical Association* 84.406 (1989-06-01). Publisher: Taylor & Francis _eprint: https://www.tandfonline.com/doi/pdf/10.1080/0 pp. 414–420. ISSN: 0162-1459. DOI: 10.1080/01621459.1989.10478785. URL: https://www.tandfonline.com/doi/abs/10.1080/01621459.1989.10478785 (visited on 2024-07-15).

[23] B. Johnson et al. “Why don’t software developers use static analysis tools to find bugs?” In: *2013 35th International Conference on Software Engineering (ICSE)*. 2013 35th International Conference on Software Engineering (ICSE). San Francisco, CA, USA: IEEE, 2013-05, pp. 672–681. ISBN: 978-1-4673-3076-3 978-1-4673-3073-2. DOI: 10.1109/ICSE.2013.6606613. URL: http://ieeexplore.ieee.org/document/6606613/ (visited on 2023-06-07).

[24] D. P. Kingma and J. Ba. *Adam: A Method for Stochastic Optimization*. 2017-01-29. DOI: 10.48550/arXiv.1412.6980. arXiv: 1412.6980[cs]. URL: http://arxiv.org/abs/1412.6980 (visited on 2024-02-28).

[25] N. Kokhlikyan et al. *Captum: A unified and generic model interpretability library for PyTorch*. 2020-09-16. DOI: 10.48550/arXiv.2009.07896. arXiv: 2009.07896[cs,stat]. URL: http://arxiv.org/abs/2009.07896 (visited on 2024-04-27).

[26] Z. Li et al. “Automating code review activities by large-scale pre-training”. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022-11-09, pp. 1035–1047. ISBN: 978-1-4503-9413-0. DOI: 10.1145/3540250.3549081. URL: https://dl.acm.org/doi/10.1145/3540250.3549081 (visited on 2024-01-26).

[27] T. Liu et al. “Understanding Long Programming Languages with Structure-Aware Sparse Attention”. In: *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR ’22. New York, NY, USA: Association for Computing Machinery, 2022-07-07, pp. 2093–2098. ISBN: 978-1-4503-8732-3. DOI: 10.1145/3477495.3531811. URL: https://dl.acm.org/doi/10.1145/3477495.3531811 (visited on 2024-06-13).

[28] F. Long and M. Rinard. “An analysis of the search spaces for generate and validate patch generation systems”. In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE ’16: 38th International Conference on Software Engineering. Austin Texas: ACM, 2016-05-14, pp. 702–713. ISBN: 978-1-4503-3900-1. DOI: 10.1145/2884781.2884872. URL: https://dl.acm.org/doi/10.1145/2884781.2884872 (visited on 2024-08-23).

[29] S. Lu et al. *CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation*. 2021-03-16. DOI: 10.48550/arXiv.2102.04664. arXiv: 2102.04664[cs]. URL: http://arxiv.org/abs/2102.04664 (visited on 2024-04-07).

[30] Maria del Pilar Angeles et al. “Analysis of String Comparison Methods During De-Duplication Process”. In: *The Seventh International Conference on Advances in Databases, Knowledge, and Data Applications*. The Second International Workshop on Large-scale Graph Storage and Management, 2015-05-24, pp. 57–62. ISBN: 978-1-61208-408-4. (Visited on 2024-07-15).

[31] D. Martens and F. Provost. “Explaining Data-Driven Document Classifications”. In: *MIS Quarterly* 38.1 (2014). Publisher: Management Information Systems Research Center, University of Minnesota, pp. 73–100. ISSN: 0276-7783. URL: https://www.jstor.org/stable/26554869 (visited on 2024-02-28).

[32] J. Nievergelt. “Exhaustive Search, Combinatorial Optimization and Enumeration: Exploring the Potential of Raw Computing Power”. In: *SOFSEM 2000: Theory and Practice of Informatics*. Ed. by V. Hlaváč, K. G. Jeffery, and J. Wiedermann. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2000, pp. 18–35. ISBN: 978-3-540-44411-4. DOI: 10.1007/3-540-44411-4_2.

[33] S.-H. Noh. “Analysis of Gradient Vanishing of RNNs and Performance Comparison”. In: *Information* 12.11 (2021-11). Number: 11 Publisher: Multidisciplinary Digital Publishing Institute, p. 442. ISSN: 2078-2489. DOI: 10.3390/info12110442. URL: https://www.mdpi.com/2078-2489/12/11/442 (visited on 2024-02-21).

[34] R. Poyiadzi et al. “FACE: Feasible and Actionable Counterfactual Explanations”. In: *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society*. AIES ’20. New York, NY, USA: Association for Computing Machinery, 2020-02-07, pp. 344–350. ISBN: 978-1-4503-7110-0. DOI: 10.1145/3375627.3375850. URL: https://dl.acm.org/doi/10.1145/3375627.3375850 (visited on 2023-06-07).

[35] M. Pradel and K. Sen. “DeepBugs: a learning approach to name-based bug detection”. In: *Proceedings of the ACM on Programming Languages*. Vol. 2. 2018-10-24, 147:1–147:25. DOI: 10.1145/3276517. URL: https://dl.acm.org/doi/10.1145/3276517 (visited on 2023-05-19).

[36] A. Radford et al. “Improving Language Understanding by Generative Pre-Training”. In: OpenAI, 2018. (Visited on 2024-02-22).

[37] C. Raffel et al. “Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer”. In: *Journal of Machine Learning Research* 21.140 (2020), pp. 1–67. ISSN: 1533-7928. URL: http://jmlr.org/papers/v21/20-074.html (visited on 2024-04-10).

[38] P. Raghavan. *Gemini image generation got it wrong. We’ll do better*. Google. 2024-02-23. URL: https://blog.google/products/gemini/gemini-image-generation-issue/ (visited on 2024-02-25).

[39] R. Russell et al. “Automated Vulnerability Detection in Source Code Using Deep Representation Learning”. In: *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA). 2018-12, pp. 757–762. DOI: 10.1109/ICMLA.2018.00120. URL: https://ieeexplore.ieee.org/abstract/document/8614145 (visited on 2024-04-07).

[40] G. Shakhnarovich, ed. *Nearest-neighbor methods in learning and vision: theory and practice*. Neural information processing series. Cambridge, Mass.: MIT Press, 2005. 252 pp. ISBN: 978-0-262-19547-8.

[41] J. Smith et al. “Questions developers ask while diagnosing potential security vulnerabilities with static analysis”. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015-08-30, pp. 248–259. ISBN: 978-1-4503-3675-8. DOI: 10.1145/2786805.2786812. URL: https://dl.acm.org/doi/10.1145/2786805.2786812 (visited on 2023-06-07).

[42] K. O. Stanley and R. Miikkulainen. “Evolving Neural Networks through Augmenting Topologies”. In: *Evolutionary Computation* 10.2 (2002-06). Conference Name: Evolutionary Computation, pp. 99–127. ISSN: 1063-6560. DOI: 10.1162/106365602320169811. URL: https://ieeexplore.ieee.org/abstract/document/6790655 (visited on 2024-02-27).

[43] J. Su, D. V. Vargas, and K. Sakurai. "One Pixel Attack for Fooling Deep Neural Networks". In: *IEEE Transactions on Evolutionary Computation* 23.5 (2019-10). Conference Name: IEEE Transactions on Evolutionary Computation, pp. 828–841. ISSN: 1941-0026. DOI: 10.1109/TEVC.2019.2890858. URL: https://ieeexplore.ieee.org/abstract/document/8601309?casa_token=Xns3uw2DYFYAAAAA:OjXke-FKOLhVN1tfOb09ucmPSWxnMCuIBwfIhEPvcGiezPT0z-7aNHiNh2U77zRyyezkdRwS (visited on 2024-02-28).

[44] H. Touvron et al. *LLaMA: Open and Efficient Foundation Language Models*. 2023-02-27. DOI: 10.48550/arXiv.2302.13971. arXiv: 2302.13971[cs]. URL: http://arxiv.org/abs/2302.13971 (visited on 2024-02-22).

[45] A. Vaswani et al. *Attention Is All You Need*. 2023-08-01. DOI: 10.48550/arXiv.1706.03762. arXiv: 1706.03762[cs]. URL: http://arxiv.org/abs/1706.03762 (visited on 2024-02-21).

[46] S. Wachter, B. Mittelstadt, and C. Russell. "Counterfactual Explanations without Opening the Black Box: Automated Decisions and the GDPR". In: *Harvard Journal of Law & Technology (Harvard JOLT)* 31 (2017), p. 841. URL: https://heinonline.org/HOL/Page?handle=hein.journals/hjlt31&id=859&div=&collection=.

[47] Y. Wang et al. *CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation*. 2021-09-02. DOI: 10.48550/arXiv.2109.00859. arXiv: 2109.00859[cs]. URL: http://arxiv.org/abs/2109.00859 (visited on 2024-04-10).

[48] Z. Yang et al. "Natural Attack for Pre-trained Models of Code". In: *Proceedings of the 44th International Conference on Software Engineering*. 2022-05-21, pp. 1482–1493. DOI: 10.1145/3510003.3510146. arXiv: 2201.08698[cs]. URL: http://arxiv.org/abs/2201.08698 (visited on 2024-06-22).

[49] Z. Zhao et al. "Knowledge-Based Version Incompatibility Detection for Deep Learning". In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023-11-30, pp. 708–719. ISBN: 9798400703270. DOI: 10.1145/3611643.3616364. URL: https://dl.acm.org/doi/10.1145/3611643.3616364 (visited on 2024-04-26).

[50] X. Zheng, C. Zhang, and P. C. Woodland. "Adapting GPT, GPT-2 and BERT Language Models for Speech Recognition". In: *2021 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*. 2021 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU). 2021-12, pp. 162–168. DOI: 10.1109/ASRU51503.2021.9688232. URL: https://ieeexplore.ieee.org/abstract/document/9688232 (visited on 2024-02-23).

[51] Y. Zheng et al. “D2A: A Dataset Built for AI-Based Vulnerability Detection Methods Using Differential Analysis”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). Madrid, ES: IEEE, 2021-05, pp. 111–120. ISBN: 978-1-66543-869-8. DOI: 10.1109/ICSE-SEIP52600.2021.00020. URL: https://ieeexplore.ieee.org/document/9402126/ (visited on 2024-04-07).

[52] Y. Zhou et al. “Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks”. In: *Advances in Neural Information Processing Systems*. Vol. 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper_files/paper/2019/hash/49265d2447bc3bbfe9e76306ce40a31f-Abstract.html (visited on 2024-08-21).

[53] Y. Zhu et al. “Aligning Books and Movies: Towards Story-Like Visual Explanations by Watching Movies and Reading Books”. In: Proceedings of the IEEE International Conference on Computer Vision. 2015, pp. 19–27. URL: https://www.cv-foundation.org/openaccess/content_iccv_2015/html/Zhu_Aligning_Books_and_ICCV_2015_paper.html (visited on 2024-02-22).