Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Identifying Distinctive Program Inputs in High Dimensional Feature Spaces

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science (M. Sc.)

eingereicht von: Konstantin Böttcher
geboren am: 12.01.1993
geboren in: Berlin

Gutachter/innen: Prof. Dr. Lars Grunske
Marc Carwehl

eingereicht am: .... verteidigt am: ....

Abstract

This thesis investigates the idea to distinguish distinct program inputs, by examining their grammatical structure. We therefore propose a novel approach to calculate distances between inputs, using a feature space derived from an input grammar, aiming to determine inputs that trigger unique program behavior. To evaluate our method, we utilized code coverage as a measure to quantify differences in program behavior and examine three python programs as test subjects. The evaluation consists of two steps. First we examine the individual relation between feature space distance and code coverage similarity. Second we compare population coverage of input sets, selected by our approach, with a baseline of random selection. While we are not able to determine more than weak correlation between distance and code coverage similarity, we find our approach to surpass the baseline, in terms of population coverage, though not by a wide margin.

Contents

1 Introduction **1**

2 Background and Related Work **3**
2.1 Input Distinctiveness 3
2.2 Input Fuzzing 5

3 Approach **7**
3.1 Configuration Grammar Mining 8
3.2 Input Vector and Feature Space 9
3.3 Normalization 10
3.3.1 Min-Max Normalization 10
3.3.2 Sigmoid Normalization 11
3.4 Dimensionality Reduction 11
3.4.1 Principle Component Analysis 12
3.5 Metric 12
3.6 Input Selection Strategy 14

4 Evaluation Methodology **15**
4.1 Feature Distance and Code Coverage 15
4.2 Population Coverage 17
4.3 Subjects 19
4.3.1 Autopep8 19
4.3.2 Pytype 22
4.3.3 Isort 23

5 Results **25**
5.1 Autopep8 25
5.1.1 Subset Selection 30
5.2 Pytype 33
5.2.1 Subset Selection 35
5.3 Isort 38
5.3.1 Subset Selection 40
5.4 Discussion 43
5.5 Threads to validity 47

6 Conclusion **48**
6.1 Future Work 49

1 Introduction

Software plays a central role in almost every aspect of today's society, from personal devices, to large-scale industrial systems and globally encompassing data networks. As software becomes more and more integrated into every critical infrastructure, ensuring its reliability, security, and performance is paramount. Failures in software can have and had devastating consequences, leading to financial loss, data breaches, and even endangering lives. The complexity of modern software systems on the other hand has grown also exponentially, making it more and more intricate to anticipate all potential points of failure. At the same time an ever growing number of users with varying skills and understanding, expects simplicity, seamless functionality and security. This demand for quality, coupled with the accelerating pace of software development, means that thorough and efficient testing in all stages of development has become a necessity. Moreover by identifying defects early, testing helps prevent costly errors in production, increases user satisfaction, and ensures that systems behave as expected in a wide variety of conditions. Software testing is therefore nowadays a core part of modern software development, torn between efficient development workflows and rigorous software analysis.

The essential goal of software testing is to uncover defects by executing a program under various conditions and ensuring it behaves as expected. Altering the conditions of a program is usually done by providing numerous different inputs. The effectiveness of testing however does not primarily depend on the number of test cases or inputs, but more importantly on the potentiality of these inputs to trigger unique program behaviour. Such *distinct* inputs can trigger irregular execution paths, expose edge cases trigger latent bugs. Identifying such inputs is therefore critical to contrive effective test suits, while on the other hand running tests with *redundant* inputs can waste time and computational resources, offering little in terms of new insights into the software's robustness.

One way of uncovering distinctiveness in program inputs is a posteriori, by examining the execution logs or program traces after the inputs have been run. However, this approach can be inefficient, requiring many inputs to be executed and analysed before finding the ones that matter or involves a lot of human labour, to derive meaningful tests manually. Moreover, by the time this information is gathered and analysed, parts of the software may have already been altered or new parts added, leaving the test cases inaccurate at best or even useless at worst. Alternatively an approach, where distinctiveness is not selected on behalf of the internal program structure, but the syntactical input space, would be less effected by changes and might therefore offer a meaningful advantage in terms of maintainability and portability of software. We call such an approach a priori, as it explores properties of test cases or inputs not after they have been run, but before.

This thesis explores one such approach of identifying distinct program inputs a priori, by assessing distinctiveness of inputs using features deduced from the inputs production grammar. As inputs need to be well-formed expressions, accepted by the program, it is clear that for every program a grammar could be derived describing the allowed input space. Whether such a grammar is in reality accessible or even formulated is another matter. Given a grammar, covering the full or just parts of the theoretical input space,

features, such as whether specific terminals had been derived, non-terminals traversed or even how long specific terms had become, could be used to determine properties of inputs. The collection of such features could furthermore be used to assess similarity of inputs for a given program. We conjecture that inputs which are different for many such features, should also be distinct in the sense that their program runs should be divergent. Therefore our approach for this thesis is to, construct a feature space by a given grammar, place inputs in this feature space, select a set of distant ones and evaluate their distinctiveness concerning their program runs. The next section will give an overview about our motivation, the research we are building on and related work, concerning the field of distance measurements and software testing. In the third section we will explain our approach in more detail, discuss some alternative methods, that can be employed and derive a formal specification for our distance measurement. Following we will introduce our evaluation methodology and justify our selection for test subjects. The evaluation will be given by subject, exploring different configurations for our approach and evaluate their effectiveness. Lastly we will discuss our results and give a prospect to open questions and potential future work to be done.

2 Background and Related Work

This chapter lays the groundwork for our approach. We will present previous research and discuss the motivation for particular choices concerning our method. We begin by exploring various methods for measuring the dissimilarity of strings. We present the feature spaces as the core concept our approach is building upon, to determine dissimilarity of inputs via spatial distances. Moreover we will illustrate difficulties arising from that approach and discuss distance metrics for such feature spaces. Furthermore we introduce the field of input fuzzing, as the framework for our evaluation and reason why we specifically employ configuration or option fuzzing, a special case of input fuzzing.

2.1 Input Distinctiveness

Fundamentally the aim of our thesis is to investigate the relation of similarity or distinctiveness of inputs and their corresponding program runs. Whereas we consider similarity in code coverage to be an appropriate measure to determine distinctiveness of program runs, it is much harder to define distinctiveness for inputs. The simplest approach would be to consider inputs simply as strings. In that case many methods could be used to calculate distances. Levenshtein distance [23] for example determines the minimum number of insertions, deletions and substitutions to derive one string from another. Another approach is based on Kolmogorov complexity [21], which is a purely theoretical concept, defining the complexity of any sequence of characters by the smallest possible program capable of producing that sequence. A computable distance measure for arbitrary sequences can be derived from that concept by using compression algorithms. Distance is thereby determined by comparing compression size of both sequences on their own and combined. A small size of the combined compressed file compared to the larger of the two individual files denotes high similarity and vice versa. A practical approach using this method is given by Feldt et al [10], who proposed a metric to measure diversity of multi-sets, using this concept of information distance derived from Kolmogorov complexity.

An alternative approach to derive distinctiveness measurements for input strings is using grammars, to determine structure in such sequences. One example of this is given by Soremekun et al. [34], who tried to find (dis-)similar inputs to given samples by deriving probabilistic grammars from those samples. A different approach we are focusing on in this thesis is inspired by a feature learner, introduced by Kampmann et al [19] as part of their tool Alhazen, which uses machine learning techniques in the form of decision trees to produce inputs and predict software behavior. The feature learner harnesses a grammar to derive features for that grammar and determines feature values for any input that is derivable from that grammar. The aggregate of all features, for a given grammar, can be considered a feature space. While we will explain the feature learner in more detail in subsection 3.2, we want to state here that features can be quantitative or qualitative, necessitating the need for normalization methods, to make distance measurements in such a mixed space feasible. How we will do this will be explained in subsection 3.3. One example for such distance measurements in mixed feature spaces is given by Ichino et al [17], who defined a generalized metric for spaces of

quantitative intervals, qualitative sets and hierarchically structured data. Our approach is similar to the proposed metric, though simpler, as we do not employ such complex data structures, but only binary and numeric features values. A definition of our metric will be given in subsection 3.5. While our feature space is simpler in its content, the feature learner, when used in its full capability tends to produce a high number of features, even for relatively simple grammars. Therefore the feature space is not only considered mixed but also high dimensional, entailing a specific problem of distance measurement, called the curse of dimensionality.

Coined originally by Richard Bellman [4], the Curse of Dimensionality states that with increasing dimensionality data points tend to get more and more remote from each other and distances tend to even out, as described by Banks et al [3]. This problem is most relevant for calculation of local distances as within clusters because these local data structures tend to disperse. Therefore much research on the field of high dimensional similarity measurement has been done in respect to the k-nearest-neighbour problem and to find local proximity in high dimensional spaces [15, 26, 31–33, 35]. A lot of research focuses thereby on the question of more efficient metrics than Euclid for high dimensional spaces. Aggarwal et al. [2] argued that the Manhattan or City-Block norm produce better results than Euclid and that fractional quasi norms lead to even better results. On the other hand Mirkes et al. [30] disputed the claim by Aggarwal et al and argue that the advantages are not consistent over different data sets and do not show significant improvements for k-nearest-neighbour. A different approach is given by Aggarwal [1] proposing only to check for proximity per dimension, by a given threshold and counting the number of similar dimensions. Hsu et al [16] discuss criteria for effective metrics in high dimensional spaces and introduce a similar metric to Aggarwal, called SDP (Shrinkage Divergence Proximity), which also uses thresholds to differentiate between close and far regions per dimension. On the other hand distance calculation of far apart points is less impacted by the curse, though as Banks et al [3] pointed out high dimensional spaces also tend to be multi-collinear, meaning dimensions are increasingly correlated with each other. This means, though our research is not concerned with local structures but with remote points, we still need to take the curse into consideration, as otherwise correlated or even redundant information might distort our measurements. We will therefore introduce in subsection 3.4 techniques to remove redundancy and reduce correlation between dimensions. We will discuss in greater detail, why we assume dimensional correlation to be of particular concern in our approach. Moreover, we will also examine the influence of different metrics on our results and examine if the claim by Aggarwal et al, that the Manhattan norm outperforms Euclid holds in our case.

As our conjecture is, that inputs that are distant in the feature space should also be distinct in program behaviour, we will employ a greedy strategy for selecting the most distant inputs consecutively. How this is implemented exactly will be explained in subsection 3.6, for now we only want to stress, that this strategy will first select the most divergent inputs and then fill up the spaces in between, which should, with increasing subset sizes lead to a more and more equally spread selection. This might be important as Chen et al. [7] argued that evenly spread test cases perform better in detecting faulty program behaviour, then completely randomly chosen ones. Therefore

even if our assumption that distant inputs are distinct does not hold, we should still find an advantage over a random selection of inputs, especially for bigger subset sizes.

In principle our approach could be used for a multitude of applications, where a grammar is given and distinctiveness of inputs is desired. For example the aforementioned Alhazen could harness our approach to reduce training data for its decision trees, by only running the most promising, distinct inputs. Another application could be the selection of seeds for search based approaches like mutation fuzzing. In this thesis however, we will focus on the capability of our approach to find sets of inputs, that cover the code base of a given program as extensive as possible. This is called population coverage and measures the aggregate code coverage of multiple program runs, triggered by a set of inputs. Because our approach is a selection algorithm that determines such a subset of inputs, we need as a basis large sets of inputs to choose from. Producing them manually would be cumbersome and since we already employ a grammar to position inputs in the feature space it is reasonable to also use it to produce these inputs. This process of automatically producing inputs for software testing is called fuzzing.

2.2 Input Fuzzing

Since its introduction by Miller et al [29] much research has been done on the field of input fuzzing. Input fuzzing is a software testing technique that generates a large volume of diverse inputs to feed into a program, with the goal of discovering bugs, security vulnerabilities, or other unintended behaviours. As described by Manes et al [25] input fuzzing, be it black- or white-box testing, can be broadly divided in two categories. On one hand there are model-less approaches, using random mutation techniques as done by [6, 22, 38]. These approaches often require seeds, which are randomly mutated, via bit flipping , block-based alterations or dictionary look ups to create new inputs, that can be tested against the program. On the other hand there are model-based approaches as [11, 24, 42], which require knowledge about the input format of the program under test to work. One such model-based approach is grammar based fuzzing, which harnesses a grammar to produce inputs, either by directly deriving inputs from the grammar or by tokenizing existing seeds and mutating those tokens, as for example done by Godefroid et al [11]. Grammar based fuzzing can be used to produce complex input data, as URL's or even input files. However, the effort of generating large, complex data is higher than with mutation fuzzing and the result space is more confined, due to its structured nature compared to model-less approaches. There is however a subdomain of input fuzzing, that is much more well suited for grammar based fuzzing approaches called configuration fuzzing. In contrast to general input fuzzing, configuration or option fuzzing is only concerned with testing optional parameters or program configurations. Input data is often times large, highly variable in structure and faults need to be handled by the program itself. On the other hand configurations are much more narrow and structurally dictated by some form of an argument parser, which also handles faulty syntax. Syntactical errors are hence not an issue of the program itself, making testing for them insignificant and therefore grammar based fuzzing more suited, than mutation based approaches.

Having said this, most configuration fuzzing research is still adopted to work with mu-

tation fuzzing approaches. This is partially due to the fact that most fuzzing benchmarks, as AFL [39] and fuzzbench [28] are designed for mutation fuzzing. Examples for such an approach are ConfigFuzz by Zhang et al [42], which harnesses manually constructed grammars to produce configurations for C programs and Li et al [24], who conducted configuration fuzzing in the field of large-scale systems, as HDFS or HBase. As our approach, however is concerned with very distinct input mutation based techniques seem unfit, as they only produce small differences from input to input. We therefore employ an approach suggested by Zeller et al [40], which consists of an *option grammar miner*, automatically extracting grammars for python programs and a random fuzzer, producing inputs by randomly traversing derivations of the grammar until no more non-terminals are left. A more detailed explanation of this grammar miner and the structure of command line options in general will be given in subsection 3.1. Creating large enough sets of such randomly fuzzed inputs, should leave us with a broad range for our approach to select the most distinct ones. This is therefore our foundation to fuzz large sets of inputs, place those inputs in the corresponding feature space and use a distance metric, to select the most distant ones. While the next chapter will go into more detail considering all steps of our approach, we will afterwords in section 4 describe our methodology for evaluation and introduce the programs we are using as test subjects. Subsequently we will present our results in section 5 and discuss our findings.

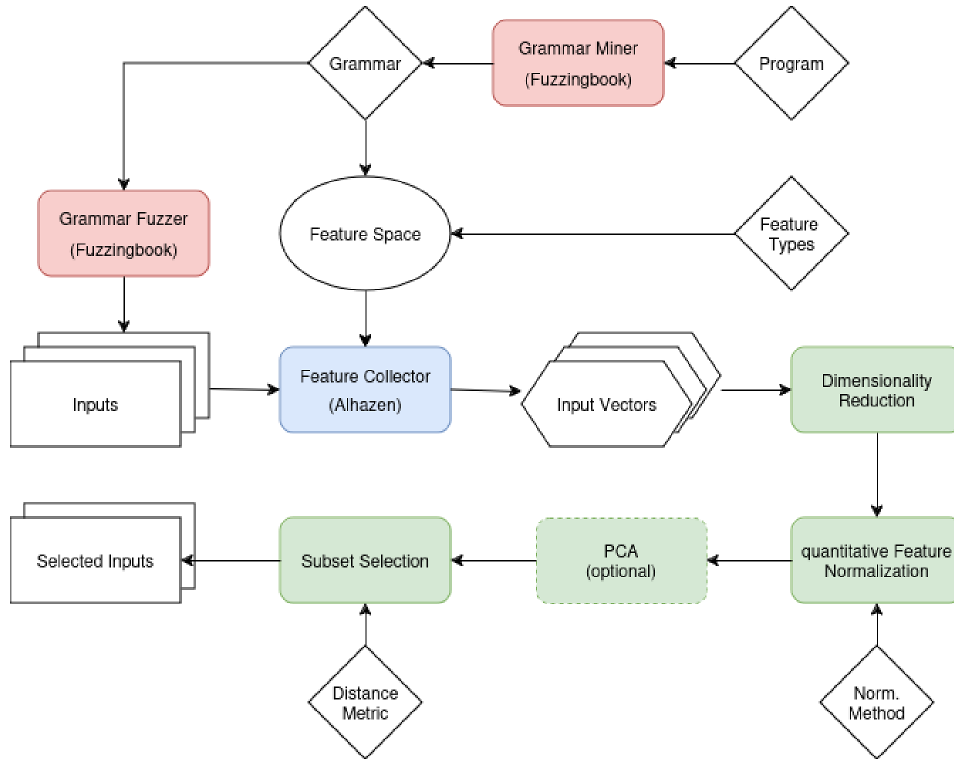


Figure 1: Subset Selection Workflow. Green shows the newly implemented components, whereas red and blue are used from previous works.

3 Approach

In this section we will give a detailed and formal account of our approach of subset selection. An overview of all steps of the method can be seen in Figure 1. As input grammars are foundational for our approach, we will start by explaining the structure of optional program parameters and how option grammars can be mined for python programs, using a grammar miner. Next we will introduce our fundamental data structure, the input vector, its feature space and how vectors can be derived from given input strings using option grammars. Building on that we will discuss multiple preprocessing steps necessary before we can perform subset selection. First we describe how and why we need to make quantitative and qualitative features comparable using normalization. Second we discuss measures for dimensionality reduction, to omit redundant information and optionally by recomposing the feature space using principle component analysis. Subsequently we can introduce our method for subset selection. We therefore will first derive the distance metric, concerning the mixed feature space and secondly explain our strategy for selection using this metric.

3.1 Configuration Grammar Mining

As already described in section 2, the prerequisite for our approach is an option grammar, to construct a corresponding feature space and determine distances for inputs. While such a grammar can be produced manually, by examining the output of the "--help" option implemented for most programs, it is also possible to mine it automatically, as demonstrated by Zeller et al in [40]. Their miner exploits the syntactical structure provided by the python argument parser *argparse*. Argparse allows to define custom command line arguments and converts the input string automatically to an object, holding all information about the defined arguments. There are two kinds of arguments: positional and optional arguments. Positional arguments are required parts of the input for the program to function. They are discernible by the fact that they are not preceded by a flag and therefore are only distinguishable by their position in the input string. This means positional arguments can't be interchanged. Optional arguments in contrast are interchangeable in position, but are always marked by a preceding flag or are indicated simply by the occurrence of a flag. They are not necessary for the function of a program, but generally modify its behaviour. A flag always has to start with a hyphen followed by an arbitrary string without spaces, however it is customary to proceed a single hyphen only by a char, while two hyphens are followed by longer strings, as in "-h" and "--help". The value following a flag is called the parameter. A typical argparse instruction to create an option has the form:

```
parser.add_argument("--flag","-f", help="description", type=bool ...)
```

Argparse allows the input string to contain multiple occurrences of the same option. In that case only the last occurrence is considered, meaning only the last parameter for an option is utilized.

To mine option grammars, Zeller et al [40] implemented an option runner, which executes a given program and records all calls of the function `parser.add_argument()`. It stops execution of the program, as soon as `parser.parse_args()` is reached, as this call indicates that no further arguments can be specified, as the input string is evaluated and transformed into a `namespace` object. At this point the option runner creates a grammar using the collected arguments, parameter names and a given base structure, that is predetermined for all option grammars, containing for example complete derivations for strings, chars and integers. As an additional input the option runner, needs predetermined values for all required or positional arguments, as otherwise most generated input strings wouldn't function. We already stated in subsection 2.2, unlike mutation fuzzing, which is useful to generate complex inputs in particularly files, the purpose of the option runner is to produce a grammar usable for fuzzing command line calls, with divers options. Therefore fuzzing of positional arguments is excluded.

Using this grammar multiple fuzzing techniques could be performed. For our application we simply employ a random fuzzer, also from the fuzzingbook by Zeller et al [41], with no adjustments other than limiting the amount of produced terminal symbols to a maximum of 100. This fuzzer will be used later on, in our evaluation, to generate sets from which

we will select distinct subsets, but that will be explained further in section 4. First we will describe our method in more detail, starting with the feature space.

3.2 Input Vector and Feature Space

In this section we will explain how we can make use of grammars to express simple input strings in high dimensional feature spaces, allowing us to compare inputs in more detail than using abstract string comparison metrics like Levenshtein [23] or Hamming distance [13]. As mentioned in subsection 2.1, the concept of deriving features from a grammar is adopted from Kampmann et all [19], but for example also employed by Eberlein et al [9].

For a given string representation of an input S_I an input vector I can be derived using a grammar $\mathcal{G}$ and a non-empty list of demanded feature types. The grammar $\mathcal{G}$ is comprised of a tuple of terminals Σ, non-terminals V, derivation rules R and a start symbol S. We only consider context free grammars, meaning derivation rules are of the form: $V \times \{V \cup \Sigma\}^*$ and expressed as $A \Rightarrow aA$. Henceforth non-terminals will be marked in rules by upper case letters and terminals by lower case letters, digits and special characters. The set of Features F can be derived a priori using the grammar. Each vector I contains numeric values for all possible features F, deducible from the grammar $\mathcal{G}$ and belonging to one of the chosen feature types. How feature values are derived for each feature depends on its feature type, of which we employ four, though more would be conceivable. We will explain next for each feature type how feature values are deduced:

- *existence feature*: deduced for each non-terminal $v \in V$ in the grammar. Can either evaluate to $true$ or $false$ depending if the term is reached when deriving the string representation of the input S_I using the grammar $\mathcal{G}$.
- *derivation feature*: deduced for each rule $r \in R$ in the grammar $\mathcal{G}$. Evaluates to $true$ if the rule was used in the derivation of S_I, otherwise evaluates to $false$.
- *length feature*: deduced for each non-terminal $v \in V$ in the grammar. Indicates the number of symbols a non-terminal represents maximal in the derivation of S_I. Consider for example term A, with the two rules: $A \Rightarrow abc$ and $A \Rightarrow vwxyz$. This term would always evaluate either to length feature value of 3 or 5. However the term B with the single rule: $B \Rightarrow bB$ can become arbitrarily long and its length feature can therefore evaluate to any natural number.
- *numeric feature*: deduced for each non-terminal $v \in V$ that represents any number and indicating that number in floating point representation. E.g. the non-terminal C with the single rule: $C \Rightarrow 7$ would always evaluate to the numeric feature value 7.0 Furthermore consider the term D with the rule: $D \Rightarrow E.E$, $E \Rightarrow EE$ and $E \Rightarrow [0-9]$, meaning E can derive any combination of digits. In this case the numeric feature for E can evaluate to any natural number and D can evaluate to any positive floating point number, even though D itself does not derive digits.

Lastly it is also possible that no number is derived from an input for a term, where there is a possible derivation for that term to a number. Since the deduction of features happens a priori there would be a numeric feature and it would evaluate to ∞. For instance the term F with the two rules: $F \Rightarrow f$ and $F \Rightarrow 1$ would have an numeric feature, but the input string "f" would evaluate for the numeric feature of F to ∞.

In the following we will consider an *input* to be the derived input vector I from its string representation S_I. We call a set of given inputs $\mathcal{I}$. An input $I \in \mathcal{I}$ is therefore a row vector of d values i_k for all features F_k, with: $k = 1, 2, ..., d$, derived from the grammar $\mathcal{G}$. Each feature F_k is a column vector containing feature values $i_{l,k}$ for all inputs I_l, with $l = 1, 2, ..., |\mathcal{I}|$, such that $F_k = (i_{1,k}, i_{2,k}, ..., i_{|\mathcal{I}|,k})$. The domain of all feature values i_k is given by U_k, thus the feature space is defined as: $U = U_1 \times U_2 \times ... \times U_d$.

Note that the existence and derivation feature types express qualitative features, while the length and numeric feature types are quantitative. As a result the former ones map always to $true$ or $false$, while the later ones map to natural and real numbers. While we can consider qualitative features as 0 or 1 and therefore obtain comparability between types, we will discuss next how to normalize quantitative features to have them range in the same interval of [0,1] as qualitative features.

3.3 Normalization

As we mentioned normalization of quantitative features is necessary for comparability between quantitative and qualitative features. Moreover it might also be beneficial with comparability within the same feature type, as it can help reduce the impact of outliers. We will introduce two different normalization techniques, namely min-max normalization and sigmoid.

3.3.1 Min-Max Normalization

The simplest method for normalization is min-max. To do so we employ min-max normalization per numeric and length feature vector, while excluding ∞ values (see subsection 3.2). Given a quantitative feature vector F, where f_{max} and f_{min} are the maximum and minimum value in that vector respectively, the normalized feature vector F^* is calculated from each value $f_l \in F$, with $l = 1, 2, ..., |\mathcal{I}|$, using the formula:

$$f_l^* = (\frac{f_l - f_{min}}{f_{max} - f_{min}}) \cdot w \tag{1}$$

The resulting normalized vector F^* contains values in the range of $[0, 1]$, while retaining proportions between values. The additional constant w is introduced to weigh quantitative features opposed to qualitative features. This is done to counterbalance the fact that only the distance between f_{max} and f_{min} will be equal to 1, while all other distances must be smaller, whereas distances in qualitative domains will always result in differences of 1

or 0. Therefore values $w > 1$ are probably more adequate for distance determination, though this has to be tested.

3.3.2 Sigmoid Normalization

An alternative approach to normalize the data is using a sigmoid function. These functions are non linear and map any real number on the interval $[-1, 1]$ without considering the whole input set $\mathcal{I}$, as min-max does. While this allows faster execution notably distance proportions are not preserved. That seems detrimental, though in some cases it might be beneficial, as it weighs values closer to zero high and values with increasing magnitude less and less. Consider inputs containing the numeric values 0.5 and 1.5, compared to inputs containing 136 and 137. Here min-max would preserve the equal distance between the pairs, though it seems more plausible that the former one results in distinct behaviour than the later one. This is captured by a sigmoid function and therefore might result in more distinct input selections than using min-max normalization. As we consider the choice of the specific sigmoid function to be of lesser importance, we use a simple sigmoid function to keep computation low and also introduce the same weighting coefficient w as with the min-max normalization:

$$\sigma(x) = \frac{x}{1 + |x|} \cdot w \tag{2}$$

3.4 Dimensionality Reduction

We discussed the curse of dimensionality in subsection 2.1. To circumvent or at least mitigate it's implications, we employ two strategies to reduce the amount of features and therefore the number of dimensions we are examining. It is reasonable to assume that the generation of features from a grammar will result in redundancy. Since input features are derived from a grammar it is likely that some features will have the same value for all inputs. We call these features *monotone*. One example for such a monotone feature is the existence of the start symbol S, which has to be by definition true for all inputs that can be derived from that language. This existence feature is evidently not meaningful for analysis. Therefore the existence feature of the start symbol S and depending on the structure of the language more monotone (existence and derivation) features are omitted from the feature space. Additionally there may also be feature pairs with the same value for all inputs. For example given a grammar, which contains exactly one rule r deriving a symbol a, it is trivial that the value of the existence feature of a and the derivation feature for the rule r will be equivalent. One of these features can be considered *redundant* and therefore is omitted without losing any information.

While this strategy will remove trivial monotone and redundant cases, there still might be highly correlated features, which might distort our distance measurements. Therefore we will furthermore test Principle Component Analysis as another technique to eliminate correlation between dimensions from our input set. Though there are more techniques to reduce dimensionality, like t-SNE [36] or UMAP [27] , these are unsuitable for our

problem, as they preserve closeness of neighbouring points or clusters, but do not maintain distance proportions of remote points, which we are mostly interested in.

3.4.1 Principle Component Analysis

Principle Component Analysis (PCA) is a method that reduces dimensionality, by transforming data points into a new vector space with fewer dimensions, still containing most of the variance from the original data. This is most beneficial in cases were disparate dimensions are correlated, as PCA can find these and reduce them, while only losing some variance.

Given a set of inputs $\mathcal{I}$ and the d dimensional feature space U^d. The goal of PCA is to transform all inputs $I \in \mathcal{I}$ from U^d into inputs $I' \in V^{d-e}$, where e is the difference in dimensions and V is a newly created feature space, with different feature domains V_k, where $k = 1, 2, ..., d - e$. To create this new feature space the principle components of the input set $\mathcal{I}$ have to be determined. The first principle component is determined by finding the line through the origin of U^d, which minimizes the error or maximizes the variance for all input vectors regarding this line. From there on, every further principle component is perpendicular to all previous components and has to maximize the variance not yet captured by any component. In total the number of principle components that will be derived is given by either the number of dimensions d or the number of inputs $|\mathcal{I}|$ minus one or: $\min(d, |\mathcal{I}| - 1)$. While the number of principle components is therefore per se not smaller than the number of dimensions, the amount of variance explained decreases drastically from component to component, meaning the last components can be dropped without loosing much information. To decide how many components to drop we simply set a cut-off point at for example 90%, meaning we only keep the first x components which together explain 90% of the variance in the original data. We will test different values for the threshold to determine how heavily the feature space can be reduced.

We expect PCA to be beneficial in our application as we assume many features to have high correlation with each other, meaning a small number of components should be able to explain a lot of variance within the data. As we already described the existence and derivation features correlate perfectly if just a single rule exists to derive a symbol. That means correlation decreases with high numbers of rules deriving a symbol, which is in many grammars uncommon for lots of symbols. Therefore derivation and existence features might be highly correlated, when only two or three rules exist. Another example would be the length and numeric feature, which also correlate for growing numbers and therefore also might be reduced efficiently by PCA.

3.5 Metric

After the data has been normalized and dimensionality reduced, input selection can happen. As our approach uses distance as the criterion for distinctiveness we first need to derive a metric in our feature space U. Therefore we first define a difference operator $\ominus$, which maps two values of the same feature domain onto a real number $(U_k, U_k) \rightarrow \mathbb{R}$. Since qualitative features map to $\{0, 1\}$, all feature types produce real numbers, meaning

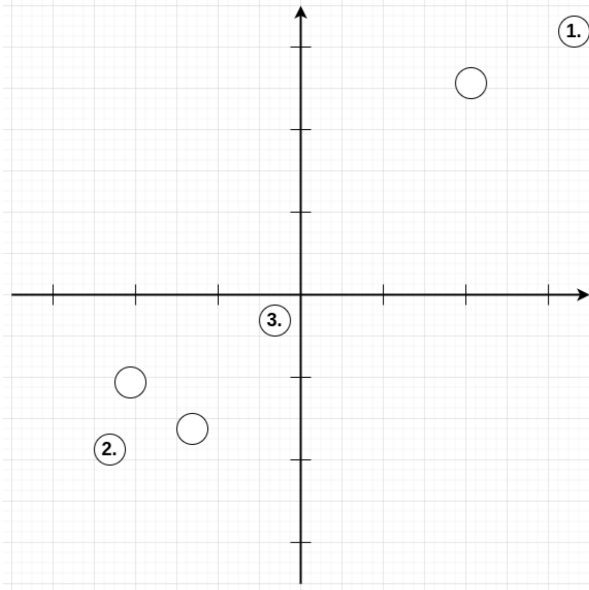


Figure 2: Example of greedy selection strategy, choosing three inputs in a 2D feature space, with numbers showing order of selection

the only exception from a normal subtraction is that a feature value for numeric features can be ∞. If either feature value is ∞ we consider the difference to be 0, as high difference is considered good. Hence the difference operator can be defined as:

$$\ominus(i_k, j_k) = \begin{cases} 0, & \text{if } i_k = \infty \vee j_k = \infty \\ i_k - j_k, & \text{otherwise} \end{cases} \tag{3}$$

with i_k, j_k being feature values in the feature domain U_k. Using this difference operator we can formulate the set of distance functions δ_p for two Inputs $I, J \in \mathcal{I}$ based on the concept of the Minkowski distance as:

$$\delta_p(I, J) = (\textstyle\sum_{k=1}^{d} | \ominus (i_k, j_k)|^p)^{1/p} \tag{4}$$

for any $p \in \mathbb{R}$, with $p \geq 1$. We employ this generalized distance metric, as it allows us to easily compare different metrics against each other. For example, choosing $p = 1$ results in the Manhattan or City-Block distance and $p = 2$ the euclidean distance. Furthermore some research showed that in high dimensional spaces choosing $p < 1$ can achieve better results [2], though this claim is disputed [30] and the result does not fulfil all criteria of a metric, as the triangle inequality does not hold. We will later evaluate different values for p and see how effective these metrics are.

3.6 Input Selection Strategy

Using the aforementioned metric we employ a greedy search algorithm to iteratively find the most distanced input to a set of already selected inputs. Figure 2 shows an example of a set of input vectors in a two dimensional feature space and a greedy selection of the three most distant points, as well as the order in which they were selected. The algorithm works as follows.

Given a set of inputs $\mathcal{I}$, a number $n \in \mathbb{N}$, with $0 < n < |\mathcal{I}|$ indicating the size of the final set of chosen distinct inputs, as well as a subset $\mathcal{D}^l \subset \mathcal{I}$ of l preselected inputs, with $0 \leq l < n$. To gather the wanted set $\mathcal{D}^n$, successively choose the input I^*, by maximizing the minimum pairwise distance for all inputs I from $\mathcal{I}$, but not already in $\mathcal{D}$, to each input J in $\mathcal{D}$ or:

$$I^* = \max_{I \in \mathcal{I} \setminus \mathcal{D}} (\min_{J \in \mathcal{D}} (\delta_p(I, J))) \tag{5}$$

using the definition of δ_p given in subsection 3.5. Append subsequently the chosen input I^* to $\mathcal{D}$, such that the new set of chosen inputs $\mathcal{D}^*$ is defined as $\mathcal{D}^* = \mathcal{D} \cup \{I^*\}$. Repeat the process with $\mathcal{D}^*$, instead of $\mathcal{D}$ until $|\mathcal{D}^*| = n$. The resulting subset $\mathcal{D}^n$ contains a combination of l preselected, as well as $n - l$ *greedily* chosen dissimilar inputs, which can than be executed with the program under question, to determine how distinct these inputs are. One exception we need to consider, is the case where $l = 0$. Since in that case $\mathcal{D} = \emptyset$, there are no inputs J and therefore no distances can be measured. Here we calculate the distance between all input vectors in $\mathcal{I}$ and the origin O of the feature space. The selected input vector I^* is the furthest distant one:

$$I^* = \max_{I \in \mathcal{I}} (\delta_p(I, O)) \tag{6}$$

from which we can repeat as usual.

We consider this greedy strategy sufficient compared to an exhaustive search strategy for two reasons. First we are selecting the input vectors which are furthest away in hopes these inputs are most distinctive, however as we already mentioned, distinctiveness becomes only apparent a posteriori, after running the program. Therefore it is not clear if an optimal solution granted by exhaustive search would even result in a more distinct outcome, compared to a greedy solution. Secondly, while greedy is already heavy in computation, an exhaustive search approach would result in much more computational complexity. Given the set of inputs $\mathcal{I}$ has size n and the required subset a of size m. The worst case complexity of distance calculations for greedy can therefore be estimated with: $\mathcal{O}(n \cdot m)$, since for each new input, meaning m times, the distance to every not yet selected input has to be calculated. On the other hand any exhaustive approach would probably calculate the distance between each pair of inputs and cache the results, resulting in $\mathcal{O}(n^2)$ only considering the amount of distance calculations. Given that m is supposed to be substantially smaller than n and bearing in mind that this approach tries to reduce compute time, it seems unlikely that such an increase in complexity, without assured advantage would be reasonable.

4 Evaluation Methodology

This section will discuss our evaluation strategy and describe the subjects we are going to use. The measure to determine distinctiveness of inputs we employ code coverage, which is a common metric employed to evaluate fuzzing techniques. As showed by Böhme et al [5], there is strong correlation between code coverage of a fuzzer and its capability to detect bugs. However they also argued that high coverage alone does not necessitates high bug detection rates, which is supported by Klees et al in [20], who examined 32 fuzzing research papers and proposed a methodology, considering code coverage only a secondary measure, after bug detection. However to our knowledge and as Zhang et al argued [42], there are currently no explicit benchmarks for configuration fuzzing, allowing to systematically evaluate bug detection capability, making code coverage the only serviceable metric for this task. To track code coverage in all following tests we employ *coverage.py*, which is a coverage measurement tool for python programs.

The evaluation will be done in two steps. First we will examine pairwise correlation of feature space distance and code coverage similarity. This will allow us to compare different configurations of our approach with each other. Second we will evaluate the capability of our approach to find a selection of inputs, which covers a programs code base most widely. Selection will be done from a large set of inputs, which is considered the gold standard. This means in contrast to the first step we will evaluate population coverage for varying selection sizes and in proportion to the gold standard. We will therefore use a assortment of configurations we examined in the first step and compare them to equally sized random selections from the gold standard, which we consider a baseline to beat. The two parts of the evaluation will be carried out using three programs: *autopep8* and *pytype* and *isort*, whose choice we will discuss. Furthermore we will describe the process of mining option grammars for these programs and discuss some customization's we made for our use case.

4.1 Feature Distance and Code Coverage

As a first step to determine if our approach is capable of finding distinctive inputs we will evaluate if high feature space distances result in dissimilar program runs. We therefore compare distance between inputs, with the similarity in code coverage in corresponding runs. The coverage $\mathcal{C}_r$ for a given run r can be considered a set of tuples $t = (f, l)$, each comprised of a file f and a line number l. Similarity of code coverage is defined using the Jaccard similarity j, also called intersection over union (IoU):

$$j(\mathcal{C}_a, \mathcal{C}_b) = \frac{|\mathcal{C}_a \cap \mathcal{C}_b|}{|\mathcal{C}_a \cup \mathcal{C}_b|} \tag{7}$$

This means, for a pair of program runs a and b, we determine the similarity j by counting the number of lines of code, which are in both coverage sets $\mathcal{C}_a$ and $\mathcal{C}_b$ and divide them by the total number of lines in any of the two coverage sets. The resulting

Jaccard coefficient is a number between 0 and 1, with 1 implying that the two sets are the same and 0 that they have no overlap.

To relate feature space distance and jaccard similarity we will randomly fuzz 2000 inputs for each of our test programs and combine them to 1000 pairs. For each of those pairs, distance is calculated via our feature space metric and similarity by intersection over union. From these two datasets pearson correlation coefficient can be derived. The pearson correlation (for samples) can be defined for a set of tuples containing two numeric values each. In our case this tuple is given by (d_i, j_i) for a set of feature space distances d and a set of jaccard similarities j, with $|d| = |j| = 1,000$. Based on that, the pearson correlation is defined as:

$$r(d,j) = \frac{\sum_{i=1}^{n}(d_i - \overline{d})(j_i - \overline{j})}{\sqrt{\sum_{i=1}^{n}(d_i - \overline{d})^2}\sqrt{\sum_{i=1}^{n}(j_i - \overline{j})^2}} \tag{8}$$

with $\overline{d}$ being the sample mean of the set d and $\overline{j}$ for j accordingly. The pearson correlation coefficient r is a value in the interval $[-1, 1]$, with 1 implying perfect correlation, 0 no correlation and -1 perfect inverse correlation. In addition to pearson correlation we will also examine the similar spearman rank correlation. While pearson correlation examines values of data points, spearman only evaluates the rank of the values within each dataset. The spearman correlation is than simply calculated using the same formula as pearson. The difference between the two is that the spearman correlation coefficient ρ shows if two datasets have monotone correlation, while pearson predicates linear relation.

Since code coverage may vary drastically for small changes in the input, while big changes in parameter values may have no influence on coverage at all, we assume that spearman correlation should be stronger than pearson correlation. Furthermore we expect code coverage and hence jaccard similarity to decrease with increasing feature space distance. Therefore we assume to find the correlation coefficients to be negative. We will use the correlation coefficients as a simple measure to compare different configurations of our approach, using different metrics, normalization methods, feature weights and employed feature types. We conjecture, that a method with high inverse correlation between feature space distance and coverage similarity will also perform good in finding distinct subsets for population coverage. Therefore we expect that strong negative pearson and spearman coefficients will indicate the best configuration for the second step in our evaluation, which is subset selection.

RQ1: Does feature space distance correlate inversely with code coverage similarity?

RQ2: Is correlation between feature space distance and code coverage similarity a predictor for high population coverage?

4.2 Population Coverage

The first step only examined pairwise relations between inputs, however our aim is to select distinctive inputs and evaluate their population coverage, which is what we will try to do in the second step of the evaluation. Given a set of program runs $\mathcal{C} = \{\mathcal{C}_1, ..., \mathcal{C}_n\}$. We define population coverage q as the amount of elements or lines of code in the union of coverages $\mathcal{C}_r$ from all runs r or:

$$q(\mathcal{C}) = \left| \bigcap_{r=1}^{n} \mathcal{C}_r \right| \tag{9}$$

Because we are not testing different input files, but only command line options it is unlikely we will be able to fully cover the code base of our test programs, even with exceedingly large population sets. Therefore we will determine sufficiently large sets of inputs, for which coverage from randomly fuzzed inputs is consistent. The coverage of these large sets is considered the gold standard, of which our approach should cover as much as possible. We will calculate distances for inputs in these sets and use our selection procedure to find the most dissimilar inputs. To assess our approach we will compare the population coverage with an equally sized set, of randomly chosen inputs from the gold standard. While the gold standard is the upper bound, this random selection can be considered the baseline to beat for our approach to have any merit. We will furthermore test different configurations of our approach against each other and random selection. Since the fuzzing of the gold standard set, but especially the random selection will lead to chance within the results, we will do 100 executions and average the results. Therefore we will also calculate 95% confidence intervals for the comparison to random selection, to see if our approach can be considered consistently better.

RQ3: Is our approach surpassing random selection in population coverage?

If our approach is capable of finding inputs that lead to distinct program traces we should see the biggest improvements in coverage at lower subset sizes. Furthermore we assume for small subset sizes, that coverage should grow very quickly with increasing size, as most inputs should yield previously unreached code. On the other hand these improvements over random should diminish for larger subset sizes and at least at a subset size equal to the size of the gold standard, our approach and random selection have to become equal. In other words while both our approach and random selection converge to the gold standard, we expect random to do so more linearly, while our approach should resemble more of a logistic curve, which first grows rapidly and then slows down. We will therefore focus our examination on subset sizes up to 20% of the gold standard, as we suppose that the changes in larger subsets should be less drastic. Moreover as our approach introduces additional compute time, it is only feasible if its reduction compared to the gold standard set size is substantial. However we will also evaluate how big our subsets have to become, before we reach complete coverage of the gold standard. As this might not happen within a subset size of 20% of the gold standard, we will also examine

for larger subset sizes how many of the 100 executions reach 100% coverage of the gold standard. While the first question discovers, if our approach is capable of outperforming the baseline with equal size, the second tries to answer if it is able to perform as good as the gold standard, with much fewer inputs.

RQ4: Can our approach achieve the same coverage as the gold standard with reasonably fewer inputs?

Lastly we want to discuss how to compare coverages of our approach to random selection. Since we measure coverages relative to the gold standard, it seems inappropriate to just calculate differences or proportions. Since it becomes harder and harder with growing coverage rates to increase any further, we propose to determine *relative improvement* not by difference in coverage, but by decrease of unreached code. Consider the following example: given two coverages with the ratios of 0.5 and 0.6 compared to the gold standard, as well as two other coverages with 0.8 and 0.9. By simply calculating the proportions, the first case would increase by 20%, whereas the second only increases by 12.5%, while the difference is obviously the same for both. We argue, that the improvement from 0.8 to 0.9 is more significant than the increase from 0.5 to 0.6, making both the simple proportion, as well as the difference unsuited as a metric. To calculate a proper measure for improvement we first derive the relative portion of unreached code, called u, for a set of coverages $\mathcal{C}_r$ in respect to its gold standard $\mathcal{C}_g$, using the population coverage function q:

$$u(\mathcal{C}_r, \mathcal{C}_g) = \frac{q(\mathcal{C}_g) - q(\mathcal{C}_r)}{q(\mathcal{C}_g)} \tag{10}$$

From that we derive our measure for relative improvement i as:

$$i(\mathcal{C}_g, \mathcal{C}_a, \mathcal{C}_b) = \frac{u(\mathcal{C}_b, \mathcal{C}_g) - u(\mathcal{C}_a, \mathcal{C}_g)}{u(\mathcal{C}_b, \mathcal{C}_g)} \tag{11}$$

with $\mathcal{C}_b$ being the random selection or baseline and $\mathcal{C}_a$ the selection of inputs from our approach. Applying this measure to our example, we find the improvement from 0.5 to 0.6 to be 20%, while the improvement from 0.8 to 0.9 is 50%. As stated, the meaning of this percentage is decrease in unreached code. Since in the first case the baseline is 0.5, meaning the missed code is $1 - 0.5 = 0.5$ and the difference is $0.6 - 0.5 = 0.1$, we reach an improvement of $0.1 \div 0.5 = 0.2$ or 20%. Where this measure becomes most advantageous is for very high coverage rates. Consider two coverages of 0.99 and 0.999. The difference and proportional increase are very small, with 0.009 and 1%, however an increase in coverage from 99% to 99.9% can be considered very significant, as the amount of unreached code is in the first case 10 times larger. Using our measure we derive an improvement of 0.9, which expresses a decrease in unreached code of 90%. On the other hand given very small coverage ratios, like 0.1 and 0.3, we find the relative improvement

to be only 22%, even though the difference is much bigger, than in the first two examples. This however, is also reasonable, given that such low coverage rates like 0.1 as a baseline should be easy to beat. Moreover, as we will show next examining our subjects, even a single program run covers more than 30% of the code. Lastly, it is obviously possible that the baseline performs better than our approach. In this case the sign will become negative and it can also happen that the value becomes smaller than -1, as for example for a baseline coverage of 0.9 and a coverage of 0.7. Here the improvement would be -2, which represents that additionally to the lines not reached by baseline, a further twofold amount of lines is also not reached by the method. While this meaning is hard to grasp, any result with a negative sign is already bad, as any method that is not capable of matching random selection has no merit for distinct subset selection.

4.3 Subjects

To carry out our evaluation, we will use three real world python programs. First, we will run our tests using *autopep8*, as a simple subject, comprised of a small code base and reasonably many command line arguments. Second, we will revise our findings with *pytype*, which is a type checker for python code and contains a much bigger code base, as well as a larger collection if command line arguments. Lastly our third choice is *isort*, which organizes import statements and was selected because of the very large amount of options it contains. A major reason for the choice of those programs is for once their widespread functionality, as well as their varying code base sizes. Secondly we chose those programs, because they require only single python files as input, while offering a wide range of command line options.

4.3.1 Autopep8

The first program we are examining for our evaluation is called *autopep8*, a style formatter for python code, released under MIT-license by Hideo Hattori [14]. Autopep8 requires a singular argument, namely a python file, which is than formatted using the pep8 styling guide [37]. We chose autopep8 because it provides 20 optional parameters, of which 12 are simple flags, making it a rather simple program for our first subject. Of the other options three require additional information in form of a number and one requires the addition of two numbers. Four more options expect strings, two of which expect pep8 error codes like E101, E901 or W391 (for a complete list and explanations see: [18]). The other two expect paths or filenames. These options are notably simply ignored, when the given string is not representing a meaningful value. Of the 20 options 9 also allow spelling in two different ways, either using a single character or the full word, e.g. `-h` or `--help`. The code base of autopep8 is comprised of 2464 lines of code all in a single file. Of these 839 lines or 34% are covered by a run with no further options given, other than the necessary input file. The test file we are using for all executions of autopep8 contains multiple violations of the style guide.

The option grammar mined by the option runner extracted a functioning grammar for autopep8. Table 1 depicts a representation of this *mined grammar*, illustrating the overall

Table 1: Autopep8 Mined Grammar

```
<start>        →  <symbol-2><arguments>
<arguments>    →  filepath/autopep8.py
<symbol-2>     →  " " | <symbol><symbol-2>
<symbol>       →  <option>
<option>       →  -h | --help | --version | -v | --verbose | -d | --diff |
                  -i |--in-place | --global-config <filename> |
                  --ignore-local-config | -r | --recursive | -j <n> |
                  --jobs <n> | -p <n> | --pep8-passes <n> |
                  -a | --aggressive | --experimental | --exclude <globs> |
                  --list-fixes | --ignore <errors> | --select <errors> |
                  --max-line-length <n> | --line-range <line> <line> |
                  --range <line> <line> | --indent-size <int> |
                  --hang-closing | --exit-code
<line>         →  <int>
<n>            →  <int>
<int>          →  <symbol-1-1><digit-1>
<symbol-1-1>   →  " " | <symbol-1>
<symbol-1>     →  -
<digit-1>      →  <digit> | <digit><digit-1>
<digit>        →  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<globs>        →  <str>
<errors>       →  <str>
<filename>     →  <str>
<str>          →  <char-1>
<char-1>       →  <char> | <char><char-1>
<char>         →  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
                  a | b | c | d | e | f | g | h | i | j | k | l | m |
                  n | o | p | q | r | s | t | u | v | w | x | y | z |
                  A | B | C | D | E | F | G | H | I | J | K | L | M |
                  N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
                  ! | " | # | $ | % | & | ' | ( | ) | * | + | , | - | . | / |
                  : | ; | < | = | > | ? | @ | [ | \ | ] | ^ | _ | { | } | ~
```

structure the miner employs to extract grammars. The generated grammar contains rules for 18 non-terminal symbols, with a total of 153 derivations. Most derivations stem from the 3 non-terminals <option>, <digit> and <char>, with all other non-terminals only derive in one way or as in four cases two ways. This means of the 18 rules, only 7 can be derived in various ways.

Using the mined grammar to produce program calls for autopep8, we ran into multiple problems trying to evaluate code coverage. First we discovered that the program would run indefinitely, when the integer value for the rule <option> → --indent-size <int> would exceed 40,000, demonstrating that changes in numeric parameters can have real world consequences in program behaviour. However since we are interested in evaluating code coverage and not fault identification, we changed the rule for <int>, such that only numbers with 3 digits would be produced, to guarantee that coverage reports for all program runs would be produced. Using this alteration we were able to complete our tests, but found that in many cases autopep8 still terminated with a failing exit code. We examined the error stream and found multiple problems with the produced inputs. First, some combinations of parameters were mutually exclusive, while others were dependent on another parameter to be set, leading to early cancellation. Since these exceptions were handled in code after argparse finished, the grammar miner could not be capable of identifying these constraints. We decided to create a second *refined grammar* next to the mined grammar, aiming to circumvent program failure. We excluded productions containing mutual exclusive options for that grammar and made sure dependent parameters were only set when feasible. Another problem was the production of integers, as no option expected negative values. We therefore deleted the rules for <symbol-1> and <symbol-1-1>from the refined grammar, leaving only positive numbers as an option. We furthermore found autopep8 to be terminating unsuccessfully, whenever the rule <option> → --line-range <line> <line> produced a smaller value for the second instance of <line>. Since we already limited all integers to a maximum of three digits, we simply changed the rule such that the second number would have one more digit: <option> → --line-range <line> <line><digit> While this does not eliminate the possibility of the error occurring, it makes it unlikely to happen, while keeping the grammar more simple, than it would be if we would have to specify rules making the second integer larger. Considering we chose this program as a contained first example, simplicity is here more important to us. After fixing these issues we found no more problems leading to early cancellation.

While the changes for the refined grammar are mostly ensuring that inputs are syntactically correct and leading to cancellation, we also wanted to see if a more semantically *optimized grammar* could outperform the mined and refined grammar. The main motivation for this step was the fact that the miner considers alternative flags of the same option as unrelated. Consider the example stated earlier of `-h` and `help`. Using the grammar given by the miner both versions would be considered different derivations, as would be the flags `-h` and `-v`, resulting in a distance of 1 in two derivation feature dimensions, though the code coverage would be the exact same. To circumvent that, we removed all single hyphen flags. Furthermore we gave every parameter a unique non-terminal symbol, from which <int> or <str> would be derived. The mined grammar uses for

some parameters of different options the same non-terminal <n>. Since <n> could stand in for number of jobs, as well as number of passes, which would be cumulated in the same feature, we wanted to see if giving them unique non-terminal symbols could improve results. The changes we made so far for the optimized grammar could be automated by a more suited option miner. However, we also revised the rules for the parameter `--errors`, which in the mined grammar would be derived to any string. This parameter is supposed to be a pep8 error code and the corresponding options would be ignored if not legal. Considering this would again mess with our distance measure we changed the rule to make it more likely an error code could be produced: <errors> → E<int> | W<int> The resulting code is not necessarily an existing error code, as not every three digit number is valid, but it should increases the chances to create one considerably.

4.3.2 Pytype

The second program we are examining is *pytype* developed by Google under the Apache 2.0 license [12]. Pytype is a static type checker for python, that analyses source code for type mismatches, without the need for manual annotation. The code base contains 31,249 lines of code spread over 187 files, making it more than 10 times larger than autopep8. The test file we used contained variables of multiple types, which are partially indicated and a function call, to force pytype making some type evaluations. Using this file we found a run without any further options specified to cover 10,867 lines or 35% of code, which is similar to autopep8. While having a much larger amount of code, the command line options are only 50% more numerous than for autopep8, with 31 arguments. Of these 21 are flags, with the other 10 requiring parameters. Two of these expect numbers, one being verbosity level and the other number of jobs. Of the other eight options four are paths, two of which are target paths for a config file and the output to write to. The other two are paths to read either a python source or a config file. Two more options are supposed to give a platform string like "linux" or "win32" and the other a python version. Lastly there are two options expecting lists of parameters, whereof one expects the lists to be comma separated, while the other `--exclude` or `-x` expects space separation. This is a strange choice, since if `--exclude` is last before the required positional argument, the required argument is considered part of `--exclude`'s list and an error is thrown.

We again produced a grammar for pytype using the miner from the fuzzingbook [40], but found the result to be deficient. For once multiple options had duplicated derivations of the form: <option> → --no-cache | --no-cache . Furthermore and even more peculiar no derivations at all for parameters where given, meaning all options were considered simple flags. We manually adjusted the grammar, in the structure of the mined autopep8 grammar and removed duplicate derivations. We furthermore needed to remove the '/' character from <char> derivations, as it appeared problematic with arguments expecting paths. For example an input included the option `--exclude /`, which resulted in exceedingly long runtime, as pytype apparently tried to traverse the whole file system. Though we also could have created alternative derivations for paths and strings, we again preferred to keep the grammar simple, to not inflate the number of features generated later on.

Fuzzing inputs using the mined grammar, we found that over 25% of program runs did not terminate successfully. We therefore again created a refined grammar, which aims to resolve runtime errors leading to early cancellation of pytype. To do so we changed the rule for <verbose> to derive 0, 1 and 2, as these are the only legal values, but also -1 to still allow for failure in some cases. Similarly we limited the derivations for <pythonVersion> to three existing ones and one incorrect. Furthermore we removed the derivation of <option> → --config <config> , as this will either always fail or if we would provide a dummy config replace the options we are trying to fuzz. Lastly we changed the derivation for <exclude>, such that there always must be another option following, as otherwise the input file was considered part of the excluded list. Using the refined grammar around 10% of runs failed, most of them because of the injected failing derivations. We did not create an optimized grammar for pytype, since we already needed to make multiple semantic adjustments to the refined grammar to prevent crashes and did not find enough possible further changes to justify another variation.

4.3.3 Isort

The third and last program we are examining is *isort*. Isort is a utility that arranges and groups import statements in python files, developed by Timothy Edmund Crosley under MIT license [8]. We chose isort as it has a wide range of command line arguments with 99 optional arguments, of which many have up to 4 different notations. 59 of these options are simple flags, six have integer parameters and the rest expect either a string, a path or in five cases specific strings, as for example in the case of `--sort-order`, which accepts only 'natural' or 'native'. The code base on the other hand is just comprised of 3,509 lines of code, which is approximately 50% bigger than autopep8, while being much smaller than pytype. Using a test file, with multiple imports we found the default input, with no options specified to reach 1,699 lines or 48% of the code base.

The derived grammar by the fuzzingbook miner was again malfunctioning. However this time there were even syntactical errors. All command line arguments were derivations of the non-terminal <group>, while <option> had no derivations, leading to an assertion error by the miner's own syntax checker. Assigning the derivations to <option> and removing the rules concerning <group> solved the syntax errors, though there still remained semantic errors. For many options the miner did not add derivations for parameters, as already happened for pytype, though this time it did for some. We added parameters for all options, that were missing them. Furthermore we restricted the size of <int> derivations to four digits, as otherwise some inputs produced runs, that would not terminate in reasonable time. Using this grammar again around 25% of runs failed. Therefore we again also created a refined grammar, analysing unsuccessful runs and adjusting the grammar to circumvent these failures. We found some mutual exclusive options, for which we created an alternative derivation. Multiple integer options did not expect negative numbers and as we already mentioned five options only accepted specific strings as inputs. We changed the derivations accordingly and removed some options that were specifically for inserting configurations. Lastly we also removed the `"/"` character from <char> derivations, as it again appeared to be problematic with some

options. The resulting grammar only failed in 7% of runs.

We mentioned isort has for many options multiple notations. As already argued for autopep8 in subsubsection 4.3.1, this might impact our results negatively. Therefore we lastly also created an optimized grammar by also removing alternative notations of the same option. This resulted in 62 alternative spellings removed for the 99 options, which should also decrease the feature space substantially.

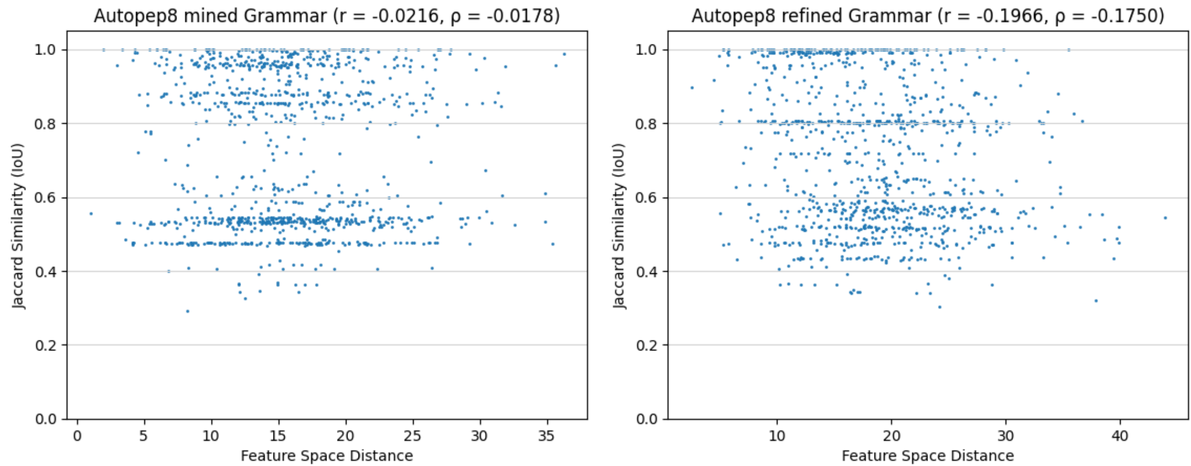


Figure 3: Scatter plots contrasting feature space distance with code coverage similarity, using the *default* configuration, with inputs from the *mined* grammar [left] and the *refined* grammar [right].

5 Results

In this section we will discuss the results for our three subjects *autopep8*, *pytype* and *isort*. We will start with autopep8, as it is the smallest program, with the most limited amount of options. Afterwords we will examine pytype and isort, using the same configurations as used for autopep8. Lastly we will discuss our results and conclude if our approach is successful in determining distinct inputs. Lastly we will name some considerations, that could limit the generality of our results.

5.1 Autopep8

We started the evaluation by comparing pairwise distance measures with code coverage similarity, as explained in subsection 4.1. First we tested the mined grammar, with the only alteration being the restriction of integers to three digits, so our tests would terminate. As a *default configuration* we decided to use the City-Block metric or $p = 1$, min-max normalization, a quantitative feature weight of $w = 1$ and not to apply PCA. We also used all four feature types to derive the feature space. Using this configuration we found the pearson correlation coefficient $r = 0,0325$ and the spearman rank correlation coefficient $\rho = 0,0321$. This means no correlation could be determined between the two measures using the mined grammar. For a visual representation we also created a scatter plot in Figure 3, showing this configuration on the left. As can be seen no clear relation between distance and similarity is visible. The smallest amount of similarity of any pair of inputs is 0.29, though the average is much higher at 0.72 and 53 pairs have identical coverage or similarity of 1.0. Considering that all runs do the same task of evaluating the identical python file, while only command line options change, this is not surprising. What is interesting, is the clustering of similarity scores around specific

Feature Types	All	Existence	Derivation	Length	Numeric
r	-0.1966	-0.2593	-0.1708	-0.2341	-0.1014
ρ	-0.1750	-0.2355	-0.1528	-0.2151	-0.0914

Table 2: Pearson correlation coefficient (r) and spearman rank correlation coefficient (ρ) for all feature types combined and every feature type alone.

values, while in the range between 0.6 and 0.8 are very few pairs. As we already argued in subsubsection 4.3.1, many inputs using the mined grammar produce errors. The clustering we are seeing is most likely the result of those errors, as all runs producing the same input error are likely to also produce the same code trace and therefore either have high overlap or very little. We examined the refined grammar (also described earlier in subsubsection 4.3.1), to see if the same clustering would appear. We found it to have better, though still low correlation scores, with a pearson correlation of $r = -0.1966$ and a spearman correlation of $\rho = -0.1750$. The corresponding scatter plot is illustrated on the right of Figure 3. As can be seen the similarity scores are more consistently spread than for the original mined grammar, though we can still recognize clustering, especially at a similarity of $j = 0.8$ and just under 0.6. Since almost all runs succeeded, we would suspect these clusters to be related to code branches that are triggered by specific options. Looking into the data we found 129 pairs that had a similarity between $[0.79, 0.81]$. Examining the inputs we found no single option that was present in the vast majority of inputs. The most frequent terminal derivation was "--diff", with 53 out of 129 occurrences for the first input of each pair and 51 for the second input. Notably the refined grammar still contains alternative notations for options, like "-d" in case of "--diff". Therefore we also counted how often "--diff" or "-d" occurred in at least one of the inputs and found the number to be 82 out of the 129 pairs in that similarity range. We also checked the other most often occurring derivations in the same way, but found their appearance to be less frequent. Since this means at least one third of pairs could not be explained by the occurrence of a single option, our assumption about the clustering being due to single important options seems to be wrong.

After this examination of specific features did not yield any clear results we wanted to evaluate next the influence of different feature types. The resulting correlation scores for specific feature types, using the refined grammar, can be seen in Table 2. There are 188 features derived from the refined grammar, however 30 are removed for being

metric (p-value)	0.1	0.5	1.0	1.5	2.0	5	10
r	-0.1080	-0.2209	-0.2570	-0.2741	-0.2837	-0.2876	-0.2499
ρ	-0.2048	-0.2232	-0.2329	-0.2389	-0.2425	-0.2469	-0.2468

Table 3: Pearson correlation coefficient (r) and spearman rank correlation coefficient (ρ) for different p-values of Minkowski distance metrics, using otherwise identical configuration.

method	weight (w)	0.1	0.5	1.0	1.5	2	10
min-max	r	-0.2554	-0.2586	-0.2570	-0.2535	-0.2497	-0.2236
	ρ	-0.2344	-0.2352	-0.2329	-0.2297	-0.2260	-0.2059
sigmoid	r	-0.2605	-0.2768	-0.2840	-0.2859	-0.2861	-0.2793
	ρ	-0.2406	-0.2512	-0.2570	-0.2599	-0.2600	-0.2580

Table 4: Pearson correlation coefficient (r) and spearman rank correlation coefficient (ρ) for different methods and weightings of quantitative features.

monotone or redundant, as described in subsection 3.4. Of the remaining 158 features, 10 are existence features, 131 derivation features, 13 length features and 4 numeric features. Examining the correlation of the different feature types isolated, we found correlation to be higher for existence and length, while derivation and especially numeric correlated poorly. Considering that the information which options are contained in the input, is only found in derivation features and not covered at all by existence features, makes this result very surprising. We would have expected, to find the best correlation with derivation features, as almost all relevant information with the given grammar should be contained by those features. An explanation might be, that the amount of derivation features is much bigger than all other feature types combined, with many of them being meaningless. Of the derivation features 93 for example only concern derivations from <char> and a further 10 derivations from <digit>, both of which are most likely not very relevant, dragging the correlation of more meaningful features down. If for example an "A" or a "5" has been derived is probably not very important on its own. To test this, we removed all derivation features concerning <char> and <digit> from our distance calculation and found the correlation for the remaining 38 derivation features to increase to $r = -0.2533$ and $-\rho = -0.2326$, with the correlation for all features, except these derivation features to increase to $r = -0.2570$ and $\rho = -0.2329$. While this improvement confirms our suspicion and shows how dependent the overall correlation is from the derivation features, it is also important to state that these correlation values are still weak. However, since the grammar miner will produce for any program that contains numbers and strings as a parameter, the rules for <char> and <digit>, we consider it feasible to remove them from the feature space for a *tweaked configuration* of our approach going forward.

We next examined if alternative metrics could improve our results. As we explained in subsection 3.5, we implemented the Minkowksy distance to test the influence of different p values, whose correlation scores can be seen in Table 3. We found a small improvement in correlation for $p = 2$ or the Euclidean distance and $p = 5$. The better correlation for distances with $p > 1$ is unexpected. Considering every dimension is of a different feature and therefore unrelated, we would think the best distance method should evaluate them independently, as City block does. An explanation could be that features are highly correlated, which would be somewhat mitigated by euclidean distance. Take for example two features f_a and f_b that are almost always the same, meaning they are highly correlated. Consider furthermore two inputs I and J, with $i_a = 1, i_b = 1$ and $j_a = 0, j_b = 0$. Since these features explain almost the same characteristic in the input,

PCA	0.5	0.8	0.9	0.95	0.99	1
r	-0.2772	-0.2233	-0.2500	-0.2488	-0.2348	-0.2280
ρ	-0.2351	-0.1825	-0.2054	-0.2115	-0.2025	-0.1956
# features	4	20	26	30	37	55

Table 5: Pearson correlation coefficient (r) and spearman rank correlation coefficient (ρ), comparing the effect of PCA for different variance thresholds, also showing the number of dimensions in the transformed space.

ideally we would not want to count them double, but only with a distance of 1. However the City block distance would evaluate their distance as $d = 2$, while euclidean would determine $d = \sqrt[2]{2} = 1.41$ and $p = 5$ would result in $\sqrt[5]{2} = 1.15$. Therefore the use of $p > 1$ might mitigate these correlation effects. However the difference in correlation is small and might therefore just be a coincidence in the produced dataset. Especially since we see the correlation drop of for more extreme values, as we would have expected.

We furthermore examined the alternative normalization method and the influence of alternative quantitative feature weights. The results can be seen in Table 4. Sigmoid normalization performed a bit better than min-max normalization, while weighting the quantitative features differently, seemed to have minuscule influence and also only decreased correlation for extreme values. A reason sigmoid performed better, might be because that method is less sensitive to outliers, though again differences are minuscule. Of the quantitative features only the numeric feature are likely to produce outliers. Therefore we checked the correlation for those features again, by using the sigmoid normalization method and found it to increase to $r = -0.1606$ and $\rho = -0.1462$, emphasizing our suspicion.

Overall we found little influence of metrics and normalization methods to improve our results. As we discussed in section 2 this could be due to the curse of dimensionality. While we already reduced the number of dimensions heavily, from 188 to 55, this is still a very high number of dimensions. Therefore we lastly wanted to see if principle component analysis could improve our approach by reducing dimensionality even further. We found that transforming the feature space by using PCA and keeping all 55 resulting dimensions, the correlation decreased to $r = -0.2280$ and $\rho = -0.1956$. This is curious, as no variance within the data is lost, but we tested it using different implementations of PCA and found the same phenomenon. The correlation improved for lower variance thresholds again, meaning fewer selected dimensions and was best when less than five dimensions were selected. This outcome suggests that many features are highly correlated with each other and is further supported by the fact that 99% of the variance is preserved by 37 dimensions in the transformed feature space, meaning 18 dimensions contain only a combined one percent of variance. Comparing different metrics after applying PCA we also found almost no difference any more, reinforcing our earlier assumption why higher p values resulted in slightly improved results.

Combining our findings and applying the determined *tweaked configuration*, using all feature types but excluding <char> and <digit> derivations, with Minkowsky metric

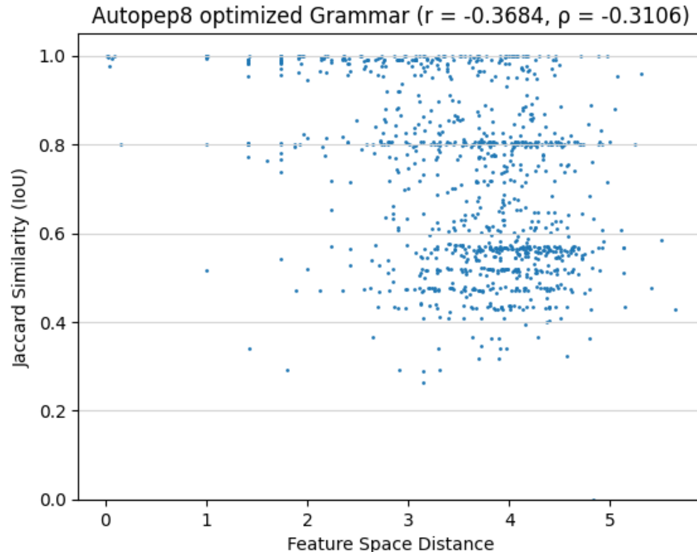


Figure 4: Scatter plot contrasting feature space distance with code coverage similarity, using the *tweaked* configuration, with inputs from the *optimized* grammar.

$p = 2$, sigmoid normalization, a quantitative feature weight of $w = 1$ and applying PCA keeping only 50% of the variance, we found the correlation to increase to $r = -0.2985$ and $\rho = -0.2630$, which is however only slightly better than our default configuration and might just be due to over optimizing the configuration for the dataset at hand. We will examine this using our other subjects in the next sections. We also applied this configuration on the mined grammar, with a resulting correlation of $r = -0.0153$ and $\rho = 0.0033$. Again we find no correlating for the mined grammar, which is most likely due to failing inputs being produced to commonly.

Lastly we described in subsubsection 4.3.1 that we also created an optimized grammar. As we already stated how important the utilized grammar is for our approach, we wanted to see if we could achieve better results, by semantically adjusting the grammar even more to our demand. Using this optimized grammar, with our default and our tweaked configuration increased the correlation to $r = -0.3253$, $\rho = -0.2926$ and $r = -0.3684$, $\rho = -0.3106$ respectively. A scatter plot using the tweaked configuration can be seen in Figure 4. While we find some improvement over the refined grammar, it is not overwhelming, with only weak correlation, applying an adjusted grammar.

Overall the results of our approach so far are not great. While we can make out some correlation it is little and only existent for manually edited grammars. It is understandable that pearson correlation is low, given the fact that inputs with small changes in distance can trigger large optional code branches or only simple conditionals, undermining the linear relation. We would however have expected to see better values with spearman rank correlation, which considers only monotone changes, but find the opposite. It has to be stated, that the difference between both correlations is small and therefore maybe just coincidence. However, it might suggest that our distance metric is less effective for near inputs, meaning their distance rank is more often mixed up in comparison to their

similarity rank, resulting in bad rank correlation. Pearson correlation on the other hand is less influenced by such small deviations as the exact values are computed. Therefore we want to state, that comparing random pairs of inputs might distort the effectiveness of our method, as our object is to find the most distinct inputs. So far no subset selection has taken place, as we only examined random input pairs. These pairs are often similar and therefore might show worse correlation, while subset selection cares about inputs with the largest distances, for which the results might be different.

5.1.1 Subset Selection

Examining the population coverage for 100 to 10,000 randomly fuzzed inputs, we found sets of 1,000 inputs to result in stable coverage sizes. For larger sizes runtime increased heavily, as coverage had to be harnessed for every input separately. Therefore 1,000 inputs are considered a sufficient size for the gold standard set, covering 56% of autopep8's code base or 1374 out of 2464 lines of code. In comparison, as stated in subsubsection 4.3.1 an input without any further option resulted in a coverage of 839 lines or 34% of code. This means for autopep8 the difference in coverage from options is 535 lines of code. We created 100 gold standard sets, consisting of 1,000 inputs each and tested subset selection on these gold standard sets. Firstly we created subsets, using our approach and random selection, increasing sizes by steps of 10, up to 200 inputs. This is done to get an overview of how population coverage increases with bigger subsets and if our approach outperforms random. Figure 5 shows a comparison of all combinations, as well as comparisons divided by grammar to their respective random selection. We illustrated comparisons with random selection per grammar, as all three produce slightly different inputs, meaning there is some difference in the coverage of random selection. We also included 95% confidence intervals, indicating consistency of these results to the baseline comparison, whereas we excluded them from the overall comparison for clarity.

Comparing the three grammars and both configurations with random selection, we find all combinations, except for the mined grammar with default configuration, to be at least on par with random selection. A special case is the subset size of 10, where no method outperformed random and most have particular bad coverages. We will examine that later on in more detail. Comparing the average coverages of the configurations with each other, differences are small and it is hard to make a conclusive decision which grammar and configuration is superior. Over all subset sizes averaged, the best is the optimized grammar with default configuration, whereas the tweaked configuration reaches consistently smaller coverage. While the difference was not large, it contradicts our correlation results, where we found the tweaked configuration to reach a higher coefficient. Moreover, this is opposite to the results for the mined grammar, where the tweaked configuration performs considerably better. The results for the refined grammar are furthermore inconclusive, with subset sizes up to 80 having higher coverage for the tweaked and onward for the default configuration. Considering we chose settings for the tweaked configuration using the refined grammar, we would have expected to see a clearer advantage, suggesting that correlation is not very predictive for population coverage.

To get a deeper insight in the differences we examined next the relative improvement,

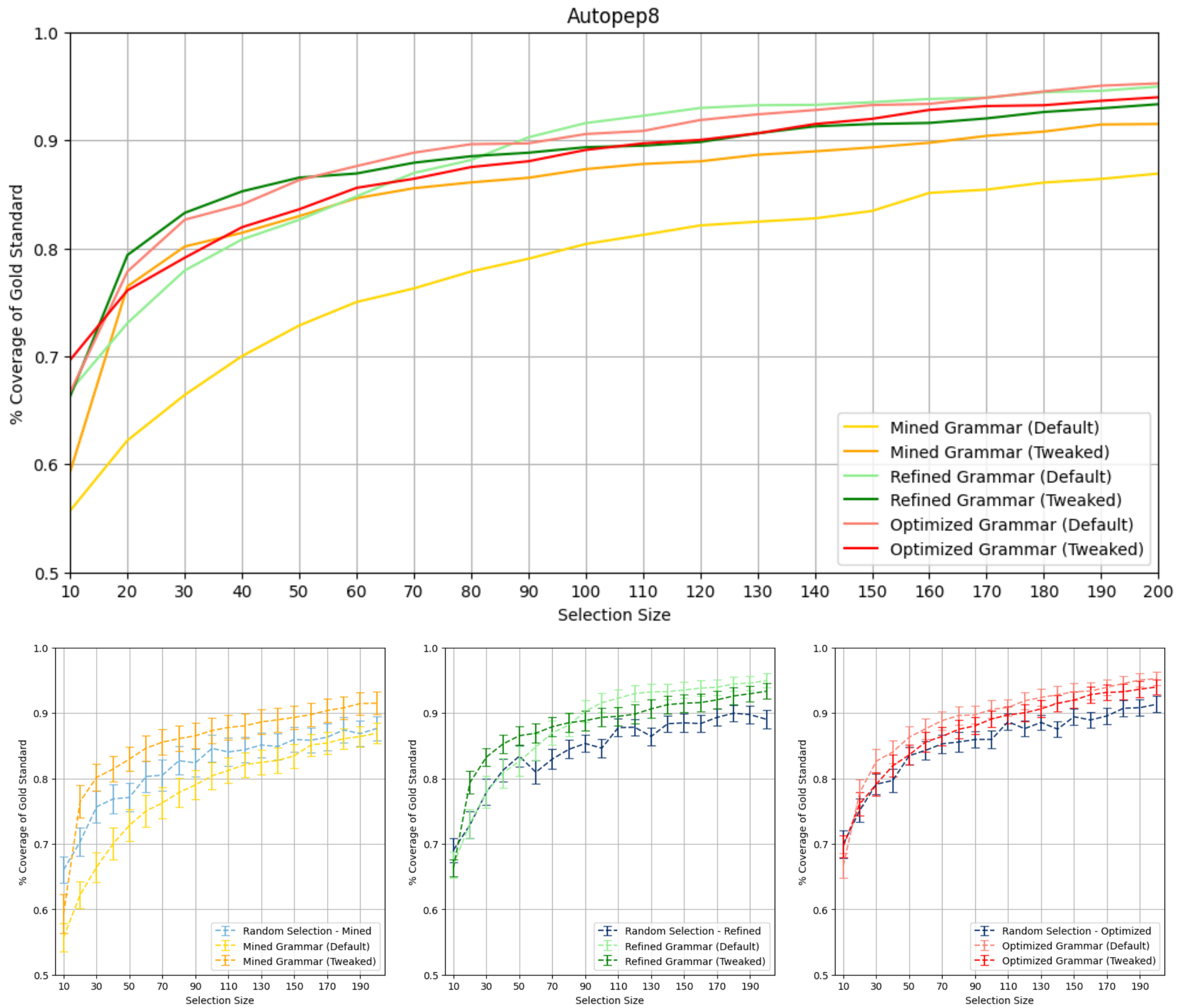


Figure 5: Code coverage in relation to gold standard for increasing subset sizes. The first plot compares different configurations and the other three show difference to random selection, with 95% confidence intervals.

we defined in subsection 4.2. The results for all combinations are shown in Figure 6. Contrary to our expectation, we find all combinations to improve more for higher subset sizes. This is due to the fact, that differences to random selection, especially over a subset size of 100, stay virtually the same, with small increases to overall coverage. As we described, our metric rates equal differences, within higher coverage, intentionally better, as it is harder to reach more code, when already having high coverage. We would not have expected the difference to stay so similar with increasing subset sizes, but suspected that random selection would catch up. Whereas on the other hand we would have expected differences to be much bigger for smaller subset sizes, as this is where a meaningful selection strategy should achieve most advantage.

Looking for example, deeper into the absolute coverage differences between random selection and refined, we find it with tweaked to get on average 3.7% more coverage. For

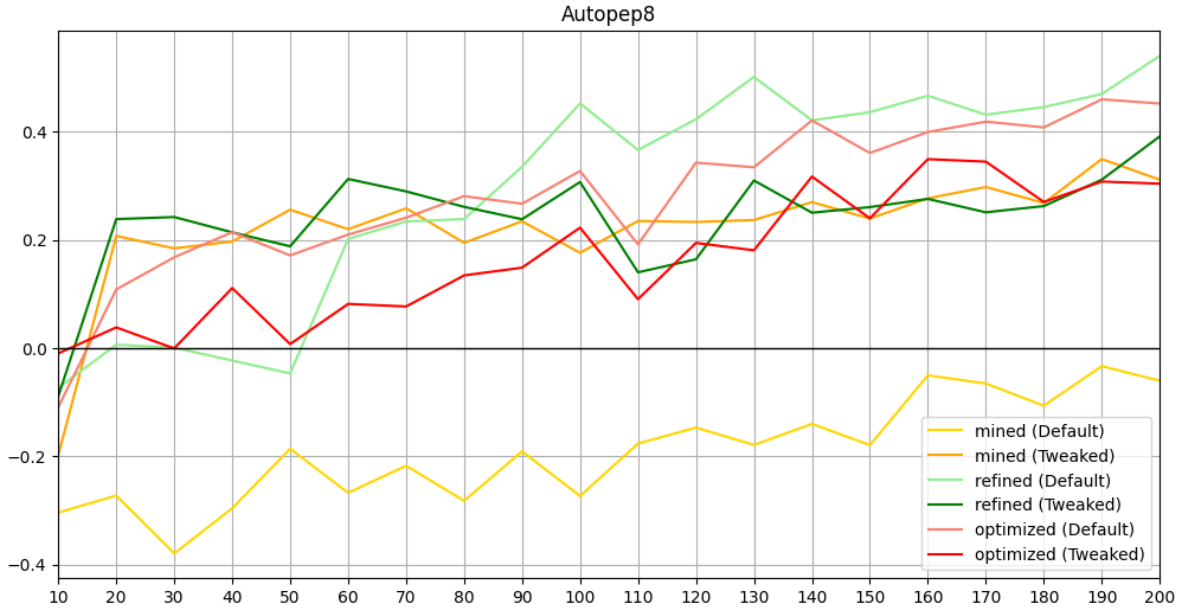


Figure 6: Relative improvement over random selection for increasing subset sizes.

the range from 10 to 100 this average is 4.6% and for 100 to 200 it is 2.9%. On the other hand using the default configuration these results flip, with the range from 10 to 100 having an average of 2.4%, while it becomes on average 5.1% above 100, with an overall average difference of 3.8%. So depending on the configuration we see vastly different behaviour. The refined grammar sees the most relative improvement, but either using the refined configuration under 80 inputs or with default configuration over 80. The most steady improvements are meanwhile seen for the mined grammar with the tweaked configuration, which excluding the special case of 10, never has an improvement lower than 0.18.

As mentioned for a subset sizes of 10 our results are special. Therefore we also analysed our approach for smaller subsets. Figure 7 shows on the left results for subset sizes of 2 to 10. We only use the tweaked configuration here, as default showed even worse results in that area. We find that the mined grammar results in significantly worse coverage than random selection for subset sizes at that range, while refined and optimized perform on par with random. The exceptionally bad performance of mined is most likely due to selection of inputs, which are not excepted by autopep8 and result in early termination and therefore small coverage. More surprising is the bad performance of the other two grammars, as we would have expected for low subset sizes to see an advantage of our approach over random, as only the most distant inputs are selected here. The fact that particularly in this area our approach performs worst is unexpected and might be due to outliers with high distance, but unexceptional traces. We therefore disabled numeric features, as these are the most likely culprit to produce outliers, run the subset selection again, but found no significant differences. We also tried other configurations, but did not find any major deviations, indicating that the most distant inputs are not of exceptional

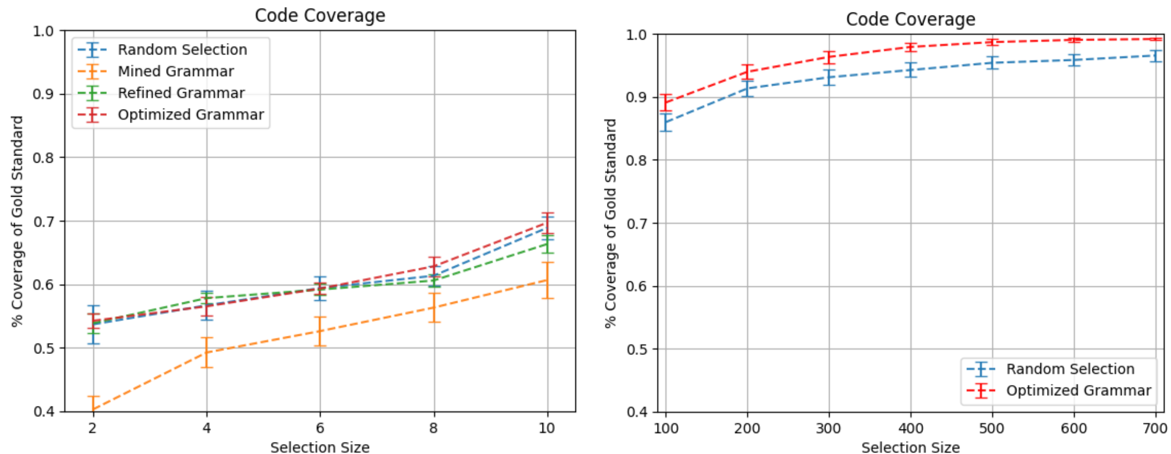


Figure 7: Code coverage in relation to gold standard, for small subset sizes [left] and very large subset sizes [right].

interest, at least for *autopep8*.

Lastly we also evaluated larger subset sizes, as we wanted to see how long it would take to reach 100% coverage of the gold standard. A depiction of the results can also be seen in Figure 7, up to a subset size of 700 inputs. At 700 we found it to reach on average 99.18% of the gold standard coverage, as well as 24 out of 100 executions, where the gold standard was completely covered. On the other hand, random reached at 700 inputs an average coverage of 96,56% and 6 executions with full coverage. While our approach performs better than random, we would have expected the results to be more levelled at such a high subset size.

Overall we find our results for *autopep8* to be underwhelming. For very small subset sizes we are not able to outperform the random selection baseline and while our approach shows better coverage for subset sizes between 20 and 200, these improvements are not huge and seemingly inconsistent. Moreover we found the differences in correlation, given by different grammars and configurations did not translate to higher population coverage. While we want to state that differences in correlation for the two configurations were not large, the population coverages using them were mostly contradictory to those scores. Lastly we can also not conclude that we reach the full coverage of the gold standard with reasonably few inputs, as even subsets with 700 inputs more often than not missed small parts of the code.

5.2 Pytype

Evaluating similarity correlation, we again started with the mined grammar using our default configuration of $p = 1$, min-max normalization, with quantitative feature weight $w = 1$, no PCA and without removing <char> and <digit> derivations. Compared to autopep8 the correlation is a bit higher though again very weak, with correlation scores of $r = -0.0848$ and $\rho = -0.0919$. Using our tweaked configuration, determined

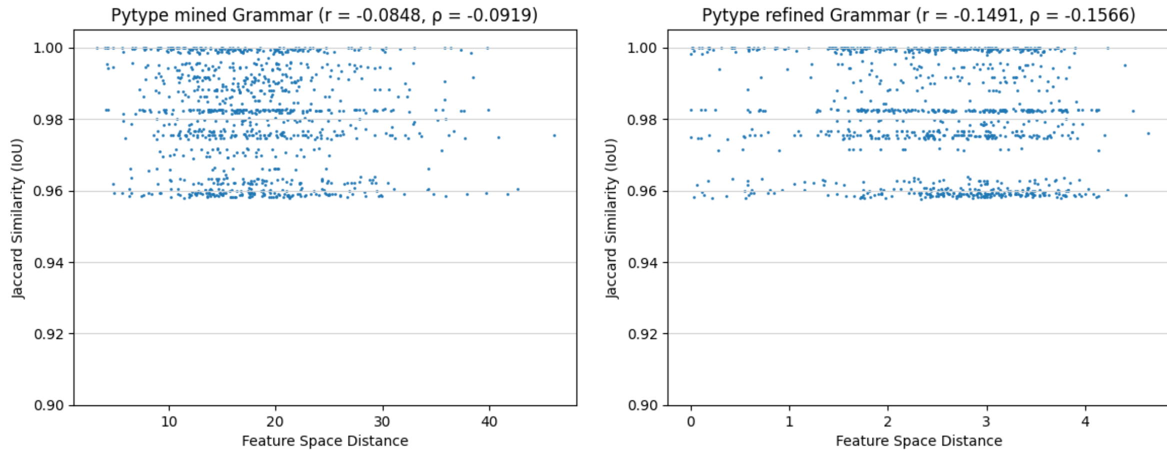


Figure 8: Scatter plots contrasting feature space distance with code coverage similarity, using the *default* configuration, with inputs from the *mined* grammar [left] and the *refined* grammar, with the *tweaked* configuration [right].

for autopep8, we find the correlation for the mined grammar to increase slightly to $r = -0.1360$ and $\rho = -0.1269$. On the other hand the refined grammar, with the default configuration gave correlations scores of $r = -0.1030$ and $\rho = -0.1096$, as well as $r = -0.1491$ and $\rho = -0.1566$ for the tweaked configuration. This means we are seeing only very small improvement over the mined grammar. We examined multiple configurations, using varying metrics, normalization methods and feature weights, but found no further improvements to the correlations scores, except a small increase by using min-max normalization, which increased the correlation coefficients for the mined grammar slightly to $r = -0.1511$ and $\rho = -0.1419$. However, since this increase is not very drastic we decided to use the same tweaked configuration for population coverage, to enhance comparability of the results.

The feature space of pytype is unsurprisingly bigger than autopep8's feature space. While the amount of options is 50% greater for pytype the feature space contains 235 or just 25% more features. Of these 47 are removed for being redundant or constant and again another 103 only consider derivations of char and digits. The reduced feature space has 85 features, compared to the 55 features of autopep8. Of these 17 are existence, 44 derivation, 19 length and 5 numeric features, which is similar in proportion to autopep8. Given that the structure of all input grammars is very similar this is not surprising. Correlation values for all feature types on their own can be seen in Table 6 and do not differ drastically from the combined score. Principle component analysis transforming and reducing the feature space even further, resulted in no improvements. Using a variance threshold of 0.5 decreased the dimensionality to 6, also very comparable to autopep8, however yielding unnoticeable increases to correlation of 0.0013 and 0.0007. While overall correlation is even weaker than for autopep8 and could not be increased drastically, we still find our tweaked configuration again to have slightly better correlation scores than the default configuration.

Feature Types	All	Existence	Derivation w/o <char>,<digit>	Length	Numeric
r	-0.1360	-0.1469	-0.1458	-0.0980	-0.0807
ρ	-0.1269	-0.1322	-0.1369	-0.0866	-0.0712

Table 6: Pearson correlation coefficient (r) and spearman rank correlation coefficient (ρ) for *pytype* (mined grammar/tweaked configuration), considering all feature types combined and every feature type alone.

The weak correlation can probably be attributed to a very high code overlap all program runs of pytype have in common. Assessing the scatter plots (see Figure 8) we found all 1,000 runs to share more than 95% of code, with the smallest similarity score being 0.9578. In absolute terms this means 10,414 lines of code were present in all 2,000 runs, with the largest coverage having 10,874 or only 460 lines more. While this number is relatively small compared to the total code base, it is in absolute terms similar to autopep8's spread in coverage and does not on its own explain such bad correlation. However an important difference is, that the default input had in autopep8's case a small coverage, whereas for pytype the default input almost covers, with 10,867 lines, as much code as the input with the most coverage. Since between the default and the maximum run lie only 7 lines of code, with an overall spread of 460 lines, suggest that in pytype's case options either lead to alternative code branches or even to skipping code, making line coverage potentially a less meaningful metric to evaluate pytype. We checked the size of the union of all lines of code in the 2,000 coverages and found it to be 10,948. This means there are only 81 lines not reached by the default input. Therefore the second case, meaning that parts of the code are skipped due to options, seems more pronounced. This also entails that a subset containing the default or alike inputs will already cover most of the code. Presumably most inputs will therefore not add anything to population coverage. We will evaluate next if our approach is capable to outperform random selection under such conditions.

5.2.1 Subset Selection

As we already discussed the default input already covers substantial part of the reachable code base. Evaluating how big the gold standard needed to be we found that 100 inputs were already sufficient to reach a stable coverage. Furthermore the high overlap in code coverage all runs share, means all methods reached the coverage of the gold standard very fast. As can be seen in Figure 9 even a single run already leads to a coverage over 95% and with 5 inputs random selection already reaches 99% coverage. Consequently the differences between configurations and grammars are slim. We see however that the default configuration performs for both grammars worse than random selection, but especially so if used with the mined grammar. On the other hand the tweaked configuration is at least on par with random selection, with only the refined grammar outperforming random selection. Though just consistently for 5 inputs and more, considering the confidence intervals.

This is also resembled when examining the relative improvements. We see in Figure 10

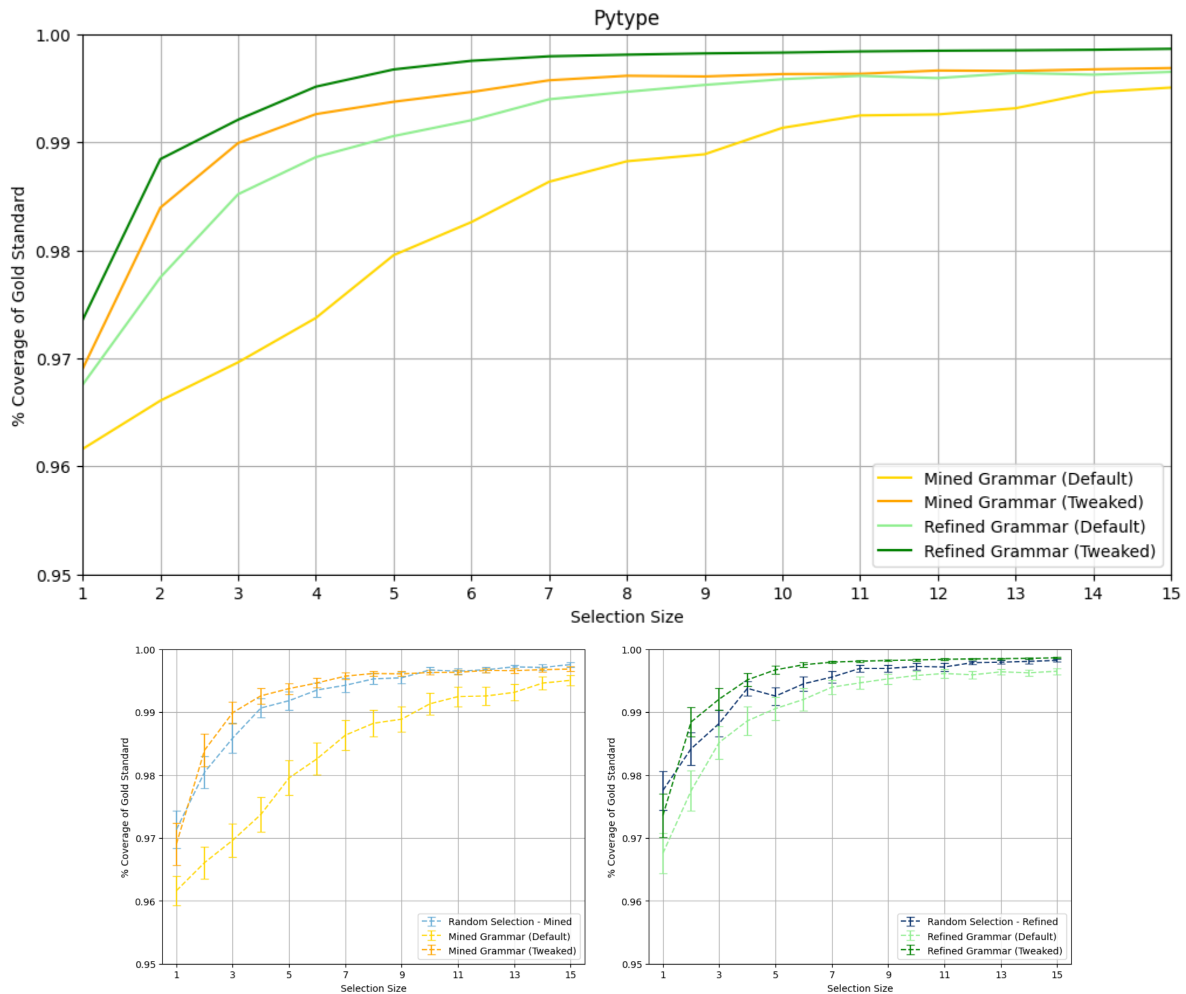


Figure 9: Code coverage in relation to gold standard for increasing subset sizes. The first plot compares different configurations and the other three show difference to random selection, with 95% confidence intervals.

that both grammars with the default configuration perform considerably worse than random selection. On the other hand the refined grammar with the tweaked configuration reaches an average improvement of 36%, with a maximum over 50%, for subset sizes between 5-7. In contrast to autopep8 we don't see the improvement to grow for larger sizes, which is due to the fact, that coverage from the refined grammar, seems to top out fast and from there on only grows minutely, while random selection still increases. For the default configuration we even found random selection to outpace it at subset sizes over 10 inputs. We also want to mention that at a subset size of 1%, random is again consistently better than any of our configurations, as was the case with autopep8. While this suggests, that our approach is not working for to small subsets, it is also noteworthy that this already changes for 2 inputs. On the other end for subsets with 15 inputs, the tweaked configuration with the refined grammar reaches an average coverage of 99.86%.

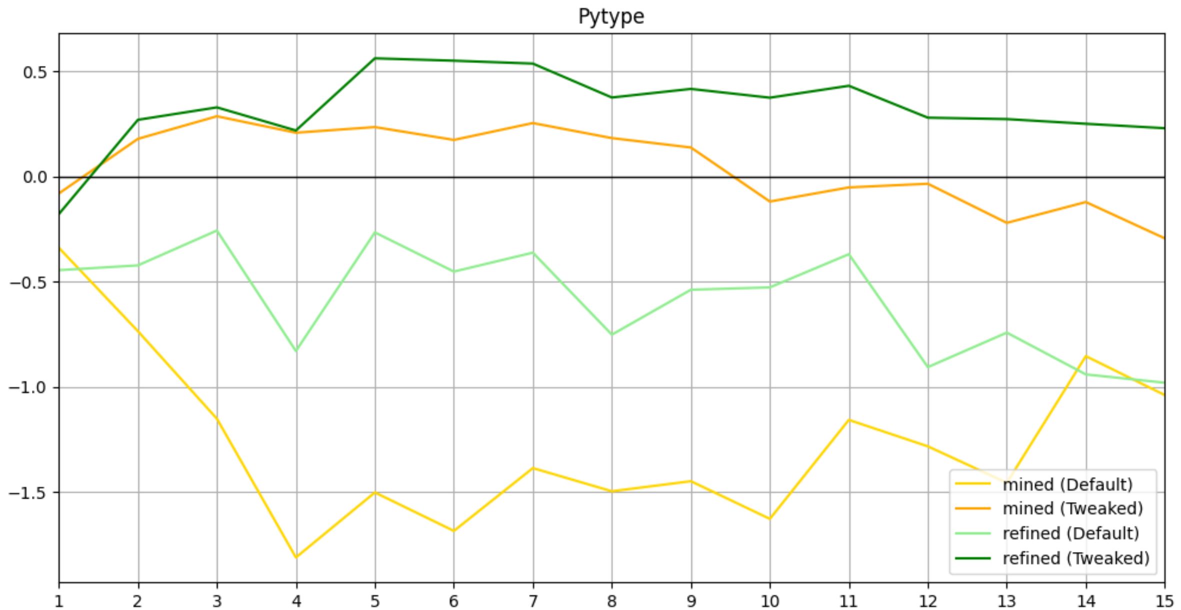


Figure 10: Relative improvement over random selection for increasing subset sizes.

However none of the 100 executions reaches perfect coverage. We therefore examined bigger subset sizes and found only above a size of 70 inputs that more than half of all executions would result in complete coverage, using the tweaked configuration with the refined grammar. Notably, random selection had at the same size only 7 executions with full coverage.

Keeping in mind, that the coverage is much higher than for autopep8, we find these results to resemble at least in part our earlier findings. Especially considering the mined grammar we see similar behaviour, with the default configuration being noticeably worse than random selection, while the tweaked configuration performs at least in part better. On the other hand considering the refined grammar, we see again better results for the tweaked configuration, compared to random, but also when comparing to the mined grammar with the tweaked configuration. The relative improvement is also overall higher compared to autopep8. However, comparing the default configuration of the refined grammar we see worse performance compared to both. So far our results suggest that our approach performs best with the tweaked configuration, but especially so, when using a manually adjusted grammar. It is not surprising that a grammar that avoids inputs that fail the program, will reach more coverage, than inputs from an unedited one. Noteworthy is that the improvement over a random selection of inputs, from the same gold standard set, is also bigger. Contrasting the population coverage results to the correlation scores, we find them overall in line, with tweaked performing better than default and the refined better than the mined grammar. Though the differences in correlation scores were so small, with the pearson coefficients differing at most by 0.0643, that no genuine conclusion can be drawn from that.

	Feature Types	All	Existence	Derivation w/o <char>, <digit>	Length	Numeric
mined	r	0.0304	-0.0276	0.0550	-0.0192	-0.0311
Grammar	ρ	0.0295	-0.0148	0.0526	-0.0059	-0.0083
refined	r	-0.0138	-0.0129	-0.0021	-0.0096	-0.0476
Grammar	ρ	-0.0025	-0.0055	0.0091	0.0070	-0.0349
optimized	r	-0.0348	-0.0058	-0.0313	-0.0222	-0.0201
Grammar	ρ	-0.1264	-0.0452	-0.1316	-0.0983	-0.0568

Table 7: Pearson correlation coefficient (r) and spearman rank correlation coefficient (ρ) for *isort* (tweaked configuration w/o PCA), considering all feature types combined and every feature type alone.

5.3 Isort

For our last program isort we again started by examining correlation scores. Considering first the mined grammar we again find no correlation, with coefficients of $r = 0.0448$ and $\rho = 0.0422$ for the default configuration, as well as $r = -0.0032$ and $\rho = 0.0047$ for the tweaked configuration. While this is similar to autopep8 and might be explained by the high rate of failure for the mined grammar, what is unexpected are equally bad scores of the refined grammar, with $r = 0.0015$ and $\rho = 0.0154$ for the default, as well as $r = -0.0052$ and $\rho = 0.0069$ for the tweaked configuration. Moreover, the optimized grammar also shows only marginally better scores for the optimized grammar with $r = -0.0603$ and $\rho = -0.1626$, already using the tweaked configuration, as the default configuration is again virtually zero.

Testing different metrics and feature weights showed no further improvements for the optimized grammar. Using min-max normalization instead of sigmoid for our tweaked configuration resulted in correlation scores of $r = -0.0590$ and $\rho = -0.1741$, meaning a minuscule increase for the spearman coefficient. While we also found a small advantage for pytype using min-max, autopep8 showed better results with sigmoid, with very small differences overall, suggesting there is no substantial difference between the two methods. All in all these values are worse than we have seen with the other two subjects, which might be due to isort having the largest compilation of options. Therefore we examined next the feature space in more detail.

For the mined grammar we found the feature space to have 343 dimensions, which was reduced by removing constant and redundant features to 300. Again around 100 of these derivations are for <char> and <digit>, leaving still 200 features or 4 times more than in case of autopep8. 171 of these are still derivation features, with only 3 numeric and 10 length features. The refined grammar on the other hand has 392 features, mostly due to an increase in derivations for options that expect specific strings. This is reduced to 254 features, so more than the mined grammar. Lastly the optimized grammar contains 325 features to begin with, which is reduced to 182, so even though we removed many alternative notations and therefore derivations of the non-terminal <option>, the feature space for the optimized grammar is still bigger than for the mined grammar. We

PCA	0.2	0.5	0.6	0.8	0.9	0.95	0.99	1
r	-0.0380	-0.0603	-0.0469	-0.0319	-0.0344	-0.0378	-0.0366	-0.0348
ρ	-0.1058	-0.1626	-0.1525	-0.1282	-0.1288	-0.1312	-0.1283	-0.1264
# features	2	7	16	50	74	89	116	182

Table 8: Pearson correlation coefficient (r) and spearman rank correlation coefficient (ρ) for *isort*, comparing the effect of PCA for different variance thresholds, also showing the number of dimensions in the transformed space.

evaluated the correlation coefficients for all feature types, using the tweaked grammar, without applying principle component analysis. Table 7 shows the results. For the mined grammar the derivation features have an especially bad correlation coefficient, as they even have a positive sign. Since these make the bulk of features the overall score is skewed by them, making the combined feature space also correlate slightly positive. Considering the tweaked configuration (where PCA is applied), the coefficients were closer to zero. This indicates that PCA reduces the influence of derivation features, suggesting once more strong dimensional correlations among the derivation features, which is reduced by PCA.

Comparing mined to refined and refined to optimized the only clear differences we see are regarding the derivation features and more pronounced with the spearman correlation. This is not surprising considering most changes to the grammar concern derivations, especially from the refined to the optimized grammar. The fact that the other features increase as well is unexpected. While the increase for length features is likely due to removing all short form notations of options, which likely weakened the correlation, because the same semantics could be expressed by shorter terminals, the reason why existence and numeric also increase is unclear. Though the difference is also minor and might just be by chance. We also tested PCA with different variance thresholds, which can be seen Table 8. The highest correlation resulted again from an aggressive variance threshold of 0.5, reducing dimensionality of the optimized grammar to 7, while smaller values started to decrease the score again. Furthermore we find by keeping 99% of variance, we already eliminate 66 dimensions or over 35% of the feature space, though do not see a change in correlation from that reduction.

Since we couldn't find clear reasons for the low correlation so far we investigated again the distribution of similarity scores. Scatter plots for the mined grammar with the default configuration, as well as the optimized grammar with the tweaked configuration are shown in Figure 11. Even though we use different grammars, the clusters we can identify are very similar in terms of similarity scores and we don't see the dispersion of them by more adjusted grammars, as we found with autopep8. In contrast to pytype similarity scores range between 0.6 and 1.0, with an average of 0.8434, 0.8980 and 0.9001 for mined, refined and optimized. The divergence in coverage is smaller, using the grammars which circumvent cancellation. This can also be seen in the plots, where there are much less points just above a similarity of 0.6 for the optimized grammar. Nonetheless comparing the scatter plot of the optimized grammar to that from autopep8's optimized grammar,

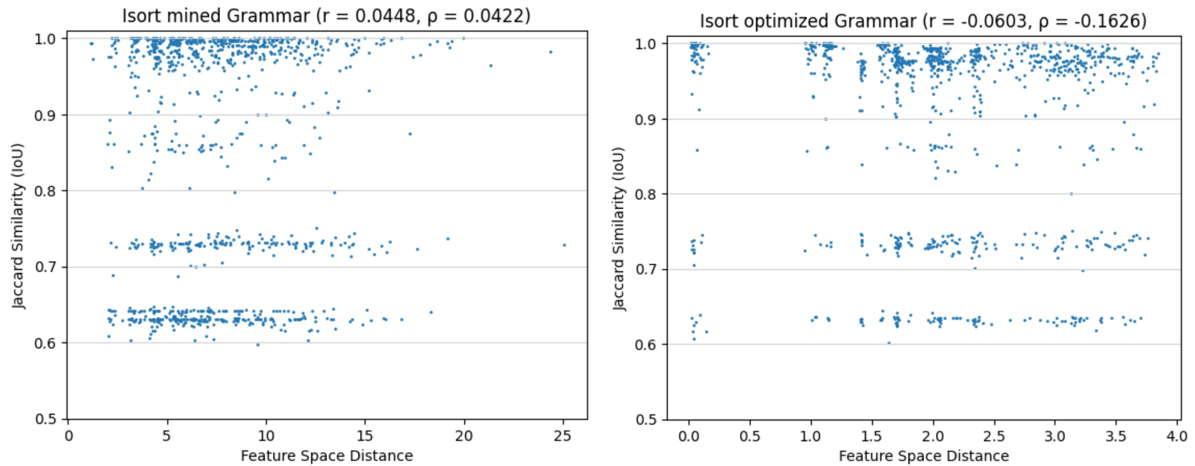


Figure 11: Scatter plots contrasting feature space distance with code coverage similarity, using the *default* configuration, with inputs from the *mined* grammar [left] and the *optimized* grammar, with the *tweaked* configuration [right].

we see much less dispersed points. High clustering around specific similarity scores leads to less pearson correlation, because no matter how different the inputs are, there seems to be just a small amount of divergent code branches, that can be taken. This might also explain the better spearman coefficient, as its rank correlation is less affected by data forming discrete steps, instead of a linear progression.

Overall however correlation is virtually non-existent. While we again find our tweaked configuration to improve scores somewhat, these differences are minuscule and a lot of manual evaluation was necessary to derive a grammar, which produced inputs rarely leading to early cancellation. On the other hand we have already seen that correlation is no clear determinant for effectiveness of population coverage, which we will examine next.

5.3.1 Subset Selection

We found again that 1,000 inputs for the gold standard size lead to stable population coverages. The gold standard covered sets covered 2300 lines of code or around 600 more than the base input. Notably however coverage.py counted 256 lines of code more for the gold standard report, due to two files from a *deprecated/* folder, that were not counted for the base case. Why coverage.py did not count those files for the base input is unclear to us, as they are evidently executable and therefore should be counted as unreached code.

Coverage is higher than in autopep8's case, as all subsets of size 10 reach already over 80% of the gold standard. Compared to pytype however this is still much lower. The results of our population coverage tests can be seen in Figure 12. All variations of our approach outperform random selection and do so consistently. We find the refined and the optimized grammar to perform virtually the same with both configurations. Furthermore,

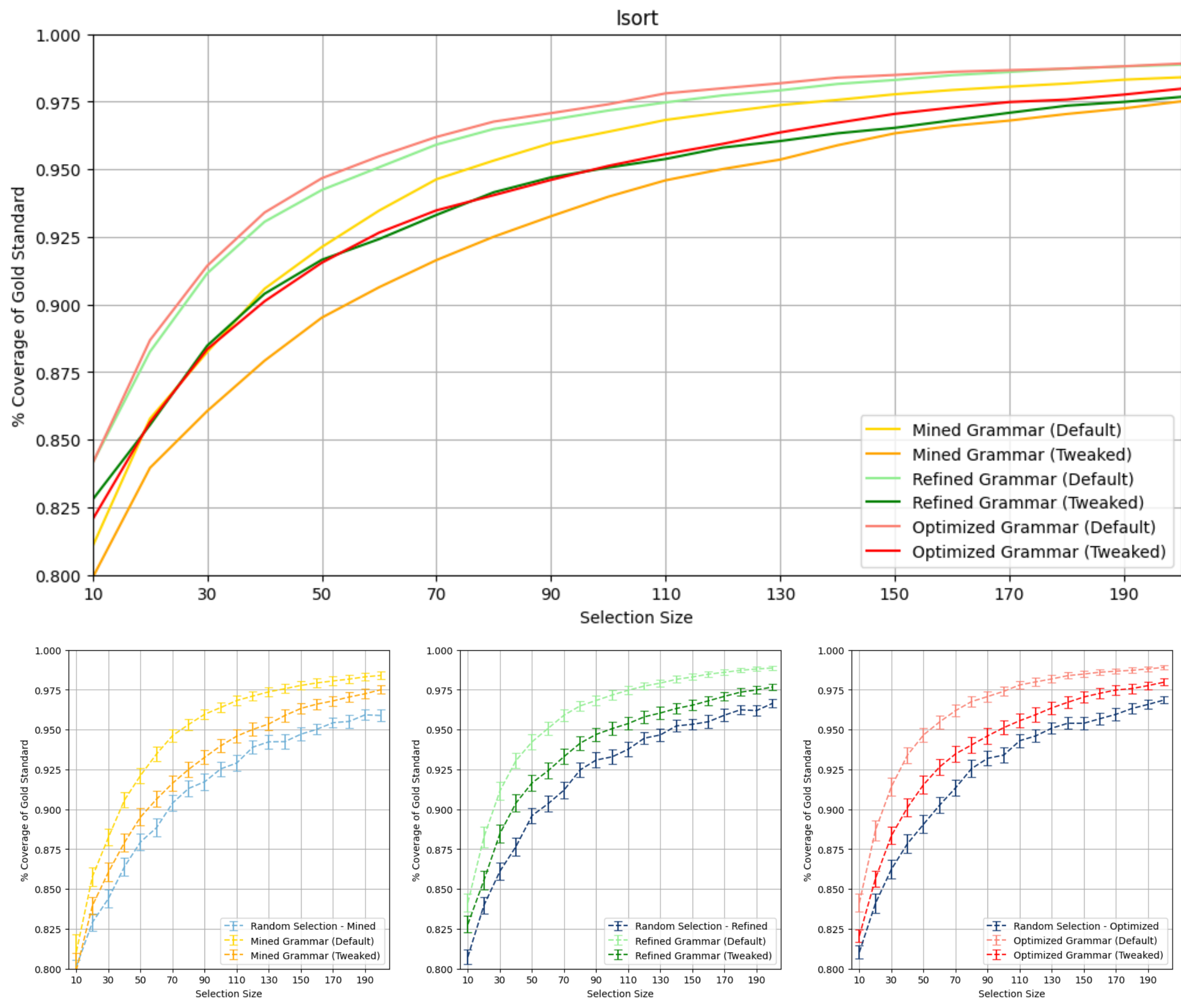


Figure 12: Code coverage in relation to gold standard for increasing subset sizes. The first plot compares different configurations and the other three show difference to random selection, with 95% confidence intervals.

we see again better coverage of inputs from those two grammars, than from the mined grammar.

However, in stark contrast to the first two subjects and even our evaluation of the correlation scores, we find the tweaked configuration to perform consistently worse than the default configuration for all three grammars. Overall we see no coherence between the correlation scores and our population coverage results for isort. Considering the correlation differences due to grammars, we saw no difference in correlation from mined to refined, though find a clear difference in coverage for both configurations. From the refined to the optimized grammar, however we saw a difference, at least for the tweaked configuration, but no difference in population coverage. Using the default configuration, with every grammar, we could not find any correlation at all, though this is the better performing one. It could be argued that the correlation coefficients where

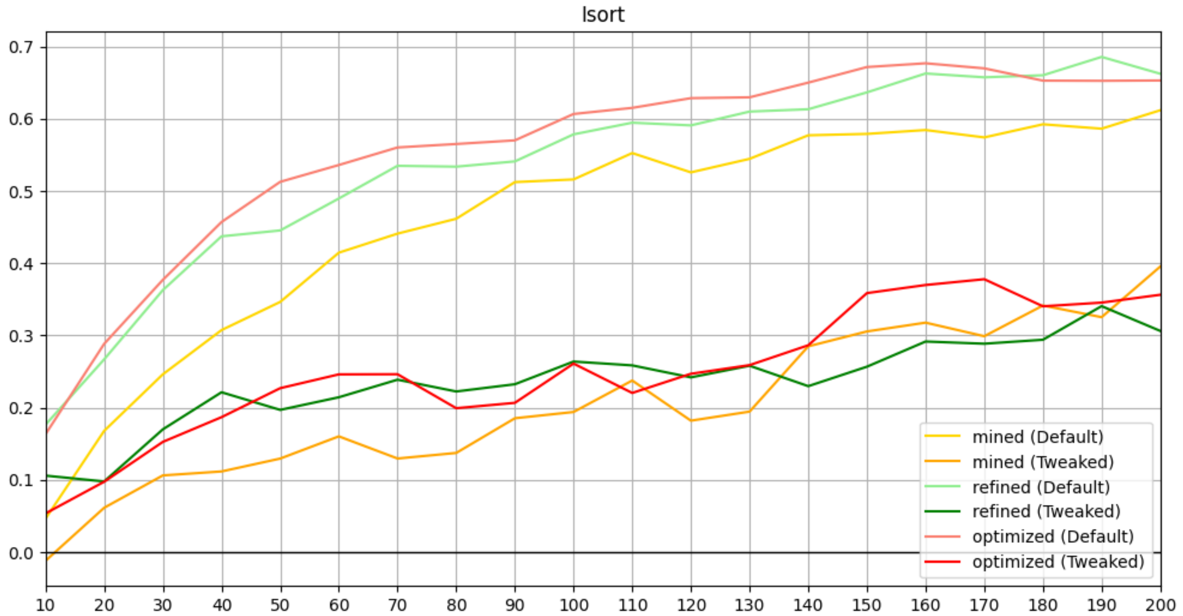


Figure 13: Relative improvement over random selection for increasing subset sizes.

extremely low no matter what configuration and therefore these differences were not meaningful. However considering how consistent our results for isort are and given the fact we already saw contradictory behaviour for the other two subjects, these findings underline that correlation is no reliable predictor for population coverage. We will discuss the implications of this in more detail in subsection 5.4, but beforehand we are going to examine the improvement of our approach over random selection, as shown in Figure 13, in more detail.

As could already be seen in the figure concerning coverage compared to random selection, the improvement is clearly divided by configuration, whereas only small differences are found concerning the grammars. While the mined grammar with the default configuration performs worse than the other two grammars with that configuration, for the first time we see it surpassing random selection and even all grammars with the tweaked configuration. As already mentioned the refined and optimized grammars reach very similar improvements over random, with the default configuration surpassing the tweaked one clearly, with an improvement that reaches almost 0.7 for subset sizes of 150 and above. We find the improvement overall to increase for bigger subset sizes, again for the same reason as with autopep8. The difference in coverage between random selection and all variations of our approach stay mostly the same, resulting in larger improvement for increasing subset sizes. We find the improvements of all three grammars with the tweaked configuration to be lower for smaller subset sizes, but overall in line with autopep8, ranging mostly between 0.1 and 0.4 and rising for larger sizes. For pytype on the other hand the resemblance is less clear. While the refined grammar with tweaked at least keeps being better than random, the same is not true for the mined grammar with tweaked, which even becomes worse than random for subset sizes over 10.

Lastly we checked again how our approach fares with much larger subset sizes, to see if it is capable of covering the whole gold standard with fewer inputs. We used the refined grammar with the tweaked configuration for a subset size of 700 and found the average coverage to be 99.84%, with 30 executions covering the gold standard completely. Employing the refined grammar with the default configuration, this increased to 99.95%, with 66 executions reaching perfect coverage. On the other hand we found random selection to reach an average coverage of 98.77%, with no execution reaching the complete gold standard coverage. Excluding the default configuration, whose very good results for isort are an anomaly, these results fall in line with our earlier findings, considering the differences in coverage for smaller subset sizes we examined. Given that autopep8 started with a smaller coverage for few inputs, the amount of executions reaching perfect coverage was also smaller, with 24 executions. Pytype on the other hand reached much higher coverage with very small subset sizes and therefore also reached 56 perfect executions, for a subset size of 70% of the gold standard size. In comparison the random selection did not manage to reach full coverage for autopep8 and isort in any execution and only in six for pytype. This means our approach is clearly surpassing random. However, considering we are talking about a subset size of more than two thirds of the gold standard, these results suggest that our approach is not capable of precisely finding all distinct inputs in considerably smaller subsets than the gold standard.

5.4 Discussion

Starting with the correlation evaluation, we found that feature space distance and code coverage similarity is at most weakly correlated and for many variations of our approach did show no correlation at all. While we found multiple cases with coefficient close to zero, we found as expected, no case where correlation was considerably positive. Against our expectation however, we found pearson and spearman coefficients to be practically equal, with only one exception for isort, where the spearman correlation was 0.1 smaller than pearson. While both coefficients indicated weak correlation at best, that shows that no stronger non-linear relation between the two variables could be found. The best correlation coefficients we found were $r = -0.3684$ and $\rho = -0.3106$ for autopep8, using the tweaked configuration and the optimized grammar, which is still a weak correlation. While we evaluated different configurations for the other subjects and still found our tweaked configuration to perform best or very close to it, autopep8 is the subject, which we examined to select our tweaked configuration, making such a low correlation result pretty conclusive.

RQ1 Answer: Feature space distance and code coverage similarity do mostly not correlate, with some exceptions of weak inverse correlation.

While correlation was overall very weak, we still found some trends comparing different configurations and grammars, for the three subjects. Using the tweaked configuration improved correlation scores, compared to default, if any inverse correlation was found

to begin with. Furthermore the refined grammar showed improvements in correlation compared to mined, as well as the optimized grammar over refined, in the two cases were we created one. While differences are overall small, we argue there is no coherence between inverse correlation of distance and similarity and increases in population coverage. As we discussed multiple times, we found cases with no real difference in population coverage, where there was difference in correlation or found difference in population coverage, where non was considering correlation.

RQ2 Answer: Correlation between feature space distance and code coverage similarity does not predict population coverage.

We can furthermore not make a clear conclusion, whether the default or tweaked configuration of our approach is superior. We found for pytype tweaked outperforming default, for isort the default being consistently superior and for autopep8 varying behaviour depending on the grammar. This suggests that the selected configuration of our approach might have little influence on the result and that advantages for specific subjects are just by chance. However, it can at least be shown that the tweaked configuration, never performed worse than random selection and using the refined or the optimized grammar always surpassed the baseline. Focusing for example on the tweaked configuration with the refined grammar, we find the averaged improvement per subject to be 31%, 36% and 25%, while covering almost always 20% more unreached code than the baseline. Though this is a relevant increase in coverage, it comes also with drawback of additional runtime. Focusing on the selection process, as the vital part of our method, to find the m best inputs from a given set of n inputs, takes $\mathcal{O}(n \cdot m)$ distance calculations, which are computationally heavy, given the high dimensionality of our feature space. While this is a substantial cost, there are plausible cases where the additional runtime by redundant inputs outweighs such preprocessing.

RQ3 Answer: Our method achieves averaged over multiple subset sizes, between 25% to 36% relative improvement in coverage over the baseline.

Unfortunately our approach is not capable of reproducing the results of the gold standard with much fewer inputs. While we found that random selection is much worse in that regard, our approach is not consistently covering the whole code, that the gold standard covered, even when selecting 70% of the inputs in the gold standard. It would be interesting to analyse how big a minimum coverage of the gold standard would have been. Our results show however, that there always were some executions, at that size, that managed to completely reach the gold standard, though not a consistent majority of them. Nonetheless, we also want to point out that random selection was clearly outperformed in that regard as well, as it basically never reached full coverage at that subset size.

RQ4 Answer: With a selection of more than two thirds of the gold standard, our method is not consistently choosing all distinct inputs.

After we answered our research questions we want to discuss some findings resulting from the evaluation process, especially concerning the grammars. As we showed, for all three subjects, inputs from a refined grammar covered more code than those from a mined grammar. While this is trivial in direct comparison, considering the refined grammars were adjusted to avoid cancellation by faulty inputs, it is noteworthy that coverage is also consistently better for those grammars in terms of relative improvement, compared to a random selection from the same pool of inputs. While we found that the mined grammar was multiple times surpassed by random selection, using a refined or optimized grammar, that was only the case for very small subset sizes of 1% of the gold standard size and less. Meanwhile it is important to state, that this outcome could be related to our evaluation methodology. Since our metric is code coverage, our result scores are tainted by program failure. Under normal circumstances, triggering and therefore finding crashes is exactly what is wanted from tests, but not so when purely measuring coverage. Therefore it can also be argued, that a low coverage for the mined grammar and hence a high rate of failure might be sought after, triggering more often faulty program states. This, though, would have to be examined in more detail, to see if the mined grammars, especially for autopep8 and pytype did in fact trigger various different failures or mostly the same. In general, however it can be stated, that with decreasing failure rate of a given grammar, the improvement in coverage over random selection grew.

In contrast we did not find that adjusting the grammar to our specific use case helped with population coverage. While we found these adaptations, resulting in the optimized grammars, to improve correlation scores, the same can not be said for population coverage, where the optimized grammars did not outperformed the refined ones. Notably our adjustment concerned primarily pruning redundant notations and did not change the overall structure of the grammar. Given the way the features are obtained, different grammars deriving the exact same language, will result in varying feature spaces. Take for example the two very similar grammars:

1: <start> → "a" | "b"

2: <start> → <A> | "b"
<A> → "a"

As can be seen the exact same language only containing the words {a, b} is derived, however one feature space will contain an existence feature for the non-terminal <A>, which the other will not have. This means within the second grammar there will be practically two features concerning the word "a", one derivation feature for "a" and one existence feature for <A>, while the first grammar has only one. While this is obviously a contrived example, it shows that the structure of the grammar has huge influence on the subsequent feature space, making the structure the grammar miner applies to produce the grammar very important. For example the grammar miner from the fuzzing book

derives options as a derivation of the non-terminal <option> in a repeating procedure, allowing the same option to be derived multiple times, as shown here:

<start> → <args><positional-args>
<args> → <option><args> | ""
<option> → ... | "-h" | "--help" | ...

An alternative grammar miner could produce options by creating a non-terminal for every option, with the derivations being the terminal or an empty string:

<start> → ...<help>...<positional-args>
<help> → "" | "-h" | "--help"

While these grammars don't produce the exact same language, both would be excepted by the program, but the resulting feature spaces would be vastly different, as the second would have an existence feature for every option, plus a derivation feature for each notation, whereas the fuzzingbook grammar has only a derivation feature for each notation. Notably it is not certain that this would be beneficial, as the number of features using that structure is much higher and the resulting inputs, using a simple random fuzzer, might be more similar. The relevant point is that the structure of the feature space could be drastically different, leading to separate distance measures. Our adjustments considering the optimized grammars only scratched the surface in that regard, however as we want to discuss lastly specific differences in feature spaces, as well as changes in the configuration of our approach might not be that important to the overall performance of the method.

After we have summed up our results, we want to finally present a reasoning for the discrepancy between the results considering individual input coverage and population coverage. As we described multiple times, results from the distance/similarity correlation, did not translate to more population coverage. We also showed that for small subset sizes random selection performed on par or even better than our approach, contradicting our assumption that the biggest improvements should be found for small subset sizes. On the other hand we found, that our approach is also not capable for large subset sizes to deliberately select all inputs resulting in more coverage. Moreover we pointed out that the optimized grammar does not outperform the refined grammar in terms of improvement of population coverage, even though its feature space should be more appropriate to determine semantically distinct inputs, which is also shown by its better correlation scores. We argue, that all of these phenomena can be explained by the conjecture, that the improvements we are seeing for population coverage stem primarily from a more equal distribution of inputs and not from a superior selection of distinct inputs.

As we already discussed, our selection strategy, after choosing the most extreme points, tends to form increasingly evenly distributed sets. This explains, why we see the worst results for very small subset sizes, as at that stage, only extreme inputs are selected, whereas even distribution becomes apparent only from a certain subset size onward. It also explains why there seems to be no relation of input dissimilarity and individual coverage to subset selection and population coverage, as it does not matter if particularly

distant points are found, just that the overall spread is even. Finally, while we reduced the amount of features in our feature space, by excluding notations of options in the optimized grammars, the produced inputs stayed semantically, meaning in terms of program behaviour the same. While that meant that individually distance was closer related to dissimilarity, the evenly distribution of inputs for population coverage did not change much, resulting in mostly the same outcome. Therefore we argue our selection strategy was capable of outperforming random, as it resulted in a more even spread of inputs. If equal spread however is the best a priori strategy to cover the largest amount of code, with a limited selection, is not clear from this thesis and must be examined further for a more definitive answer.

5.5 Threads to validity

Several factors could potentially affect the generality and reliability of the results obtained in this thesis. First, only three programs were evaluated, which limits the statistical power of the analysis. While the selected programs were chosen to represent disparate purposes and cover variably sized code bases and configurations, a broader set of subjects might yield more robust conclusions.

Secondly our assessment for the effectiveness of our approach was only compared against a baseline of randomly chosen inputs, but not against more proficient competing methods. Random selection as a baseline is an important point of comparison, as it shows if there is efficacy to our approach at all. However including additional techniques for selecting subsets could have offered more insight into the relative performance of the proposed method, by examining how easy or hard it is to beat the baseline.

Lastly we also can not be certain that increasing the gold standard sizes much more drastically would not have changed results. While we found coverage to stabilize at different points for different subjects, possibly with much bigger sets the random fuzzer might have produced much more complex inputs, which could have reached otherwise uncharted areas of the code. Besides it could also be that our approach starts to behave differently when the gold standard size is increased substantially, as more candidates emerge and the high dimensional space is populated more densely.

6 Conclusion

In this thesis we presented an approach to find distinct program inputs, using feature spaces derived from input grammars. Our objective was to determine if inputs, that are distant in this feature space do also entail different program behaviour and can therefore be considered distinct. We developed and implemented a workflow to test this hypothesis and used configuration fuzzing, as a framework to evaluate it. Using a grammar miner we generated configuration grammars for three different python subjects, namely autopep8, pytype and isort, to examine the effectiveness of our approach.

The evaluation was carried out with different configurations of our approach and examined the influence of varying input grammars. It followed a two-step process. First we analysed the correlation between the distance of individual program inputs within the feature space and the similarity of associated code coverage sets. Second, we assessed population coverage by comparing subsets selected by our approach from a gold standard set, against equally sized random selections from the same gold standard. While the first part of our evaluation was done to give a general overview if feature space distance is related to unique program behaviour, the second phase gave us an actual assessment of effectiveness of our approach, with an theoretical upper bound, being the gold standard and practical lower bound, in the form of random selection. This means, whereas the second part of our evaluation was done to determine utility of our method, the first concerned the question if improvements we are seeing are actually resulting from distinctiveness.

Our results showed no consistent correlation between feature space distance and code coverage similarity. This lack of strong correlation persisted across various grammars, subjects and examining both pearson and spearman correlation, suggesting that distance in feature space does not imply distinct program behaviour. We also tried to adjust our method, by testing different distance metrics, normalization methods and dimensionality reduction methods, however did not find strong increases in correlation. Furthermore, we were not able to predict population coverage improvements, from changes in correlation coefficients.

Nevertheless, our method achieved, with one configuration, consistently better population coverage results than the baseline. We found inconsistent behaviour for grammars generated by the miner, though these grammars also produced many inputs that failed the program, obfuscating the coverage results. Using refined grammars, circumventing most of these failures, alleviated these problems and lead to more consistent improvements. In general, however the advantage our method attained over the baseline was not vast and did not came apparent, contrary to our expectation, for very few subset sizes. Moreover, we also found our approach did not consistently accomplish to find all distinct inputs, for very large subset sizes, bigger than two thirds of the gold standard.

We discussed how different ways of constructing the input grammar might impact the outcome, by resulting in vastly different feature spaces. However, considering the discrepancies between the correlation and the population coverage results, we argued that the improvements we found, concerning population coverage, are most likely not due to a relation between feature space distance and distinctiveness, but a more equally

distributed selection of inputs, compared to random selection. Therefore we claim, that it is not distinctiveness in such a feature space that results in high code coverage, but simply an even distribution of those inputs.

6.1 Future Work

Many open issues have already been stated throughout the thesis. We want to focus first on the comparably small amount of research on the field of configuration fuzzing. As already mentioned, to our knowledge, there are no explicit benchmarks in the field of configuration fuzzing, providing test subjects for bug detection. Development of such benchmarks, next to the existing mutation benchmarks, would be of great help in terms of comparability, for research that is concerned with grammar fuzzing approaches.

With regards to our approach, various aspects could be examined in greater detail. As we suspect our improvements to stem likely from equally distributed inputs, it would be most interesting to investigate that claim further. To do so, an alternative selection strategy could be implemented, that simply chooses a subset of equally distributed inputs and compare if the coverage is the same or even better. This approach could use the same feature space and distance methods, however it might also be insightful to see if results stay the same if the input space is determined in a different way, which would heavily reaffirm the claim that it is mostly equal distribution, that resulted in improved coverage over random selection. Moreover, no comparison to alternative approaches for input selection was performed in our thesis. Therefore a comparison to other methods would be very interesting, to see how good our approach fares. This could range from comparing to generic string distance metric as the aforementioned Levensthein distance, to collecting input sets already of the subset size, using alternative fuzzing techniques and comparing them to our subsets. Moreover an a posteriori evaluation of optimum coverage sets, for given subset sizes, could also be informative, to contrast how much redundancy is still contained in our subset selections and how many inputs are minimally needed to cover the gold standard.

Lastly, building on our work, it could also be evaluated if subsets determined by our approach could be valuable seeds for mutation fuzzing approaches. While we couldn't find great results when using our inputs directly, it might be that there grammatical distinctiveness from another, constitutes a solid basis for mutation based approaches. On the other hand it could also be evaluated, if our approach could be beneficial in reducing redundant data for machine learning approaches as Alhazen, from which we adopted the feature space learner.

References

[1] C. C. Aggarwal. Re-designing distance functions and distance-based applications for high dimensional data. *ACM Sigmod Record*, 30(1):13–18, 2001.

[2] C. C. Aggarwal, A. Hinneburg, and D. A. Keim. On the surprising behavior of distance metrics in high dimensional space. In *Database Theory—ICDT 2001: 8th International Conference London, UK, January 4–6, 2001 Proceedings 8*, pages 420–434. Springer, 2001.

[3] D. L. Banks and S. E. Fienberg. Data mining, statistics. In R. A. Meyers, editor, *Encyclopedia of Physical Science and Technology (Third Edition)*, pages 247–261. Academic Press, New York, third edition edition, 2003.

[4] R. Bellman. *Adaptive Control Processes: A Guided Tour.* Princeton University Press, 1961.

[5] M. Böhme, L. Szekeres, and J. Metzman. On the reliability of coverage-based fuzzer benchmarking. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1621–1633, 2022.

[6] S. K. Cha, M. Woo, and D. Brumley. Program-adaptive mutational fuzzing. In *2015 IEEE Symposium on Security and Privacy*, pages 725–741. IEEE, 2015.

[7] T. Y. Chen, H. Leung, and I. K. Mak. Adaptive random testing. In *Advances in Computer Science-ASIAN 2004. Higher-Level Decision Making: 9th Asian Computing Science Conference. Dedicated to Jean-Louis Lassez on the Occasion of His 5th Birthday. Chiang Mai, Thailand, December 8-10, 2004. Proceedings 9*, pages 320–329. Springer, 2005.

[8] T. E. Crosley. isort. `https://github.com/PyCQA/isort`. Accessed: 2024-10-14, Commit SHA: 7de182933fd50e04a7c47cc8be75a6547754b19c.

[9] M. Eberlein, M. Smytzek, D. Steinhöfel, L. Grunske, and A. Zeller. Semantic debugging. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 438–449, 2023.

[10] R. Feldt, S. Poulding, D. Clark, and S. Yoo. Test set diameter: Quantifying the diversity of sets of test cases. In *2016 IEEE international conference on software testing, verification and validation (ICST)*, pages 223–233. IEEE, 2016.

[11] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation*, pages 206–215, 2008.

[12] Google. pytype. `https://github.com/google/pytype`. Accessed: 2024-10-03, Commit SHA: 83e28a715021d9cbd442ce4e2b33965212d58028.

[13] R. W. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 29(2):147–160, 1950.

[14] H. Hattori. autopep8. `https://github.com/hhatto/autopep8`. Accessed: 2024-08-23, Commit SHA: 916d6998765aca5de8370ce9736d56fefb3858f1.

[15] A. Hinneburg, C. C. Aggarwal, and D. A. Keim. What is the nearest neighbor in high dimensional spaces? In *26th Internat. Conference on Very Large Databases*, pages 506–515, 2000.

[16] C.-M. Hsu and M.-S. Chen. On the design and applicability of distance functions in high-dimensional data space. *IEEE Transactions on Knowledge and Data Engineering*, 21(4):523–536, 2008.

[17] M. Ichino and H. Yaguchi. Generalized minkowski metrics for mixed feature-type data analysis. *IEEE Transactions on Systems, Man, and Cybernetics*, 24(4):698–708, 1994.

[18] I. L. Johann C. Rocholl, Florent Xicluna. Pep8 error codes. `https://pycodestyle.pycqa.org/en/latest/intro.html#error-codes`. Accessed: 2024-08-25.

[19] A. Kampmann, N. Havrikov, E. O. Soremekun, and A. Zeller. When does my program do this? learning circumstances of software behavior. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 1228–1239, New York, NY, USA, 2020. Association for Computing Machinery.

[20] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 2123–2138, 2018.

[21] A. N. Kolmogorov. On tables of random numbers. *Sankhyā: The Indian Journal of Statistics, Series A (1961-2002)*, 25(4):369–376, 1963.

[22] C. Lemieux and K. Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, pages 475–485, 2018.

[23] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[24] J. Li, S. Li, K. Li, F. Luo, H. Yu, S. Li, and X. Li. Ecfuzz: Effective configuration fuzzing for large-scale systems. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12, 2024.

[25] V. J. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. Fuzzing: Art, science, and engineering. *arXiv preprint arXiv:1812.00140*, 2018.

[26] D. Marshall. Nearest neighbour searching in high dimensional metric space. *in-Department of Computer Sciences. vol. Master Thesis in Information Technology: Australian National University*, 2006.

[27] L. McInnes, J. Healy, and J. Melville. Umap: Uniform manifold approximation and projection for dimension reduction. *arXiv preprint arXiv:1802.03426*, 2018.

[28] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya. Fuzzbench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 1393–1403, New York, NY, USA, 2021. Association for Computing Machinery.

[29] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[30] E. M. Mirkes, J. Allohibi, and A. Gorban. Fractional norms and quasinorms do not help to overcome the curse of dimensionality. *Entropy*, 22(10):1105, 2020.

[31] M. Muja and D. G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *IEEE transactions on pattern analysis and machine intelligence*, 36(11):2227–2240, 2014.

[32] Rodrigues. Combining minkowski and chebyshev: New distance proposal and survey of distance metrics using k-nearest neighbours classifier. *Pattern Recognition Letters*, 110:66–71, 2018.

[33] V. Singh and A. K. Singh. Simp: accurate and efficient near neighbor search in high dimensional spaces. In *Proceedings of the 15th International Conference on Extending Database Technology*, pages 492–503, 2012.

[34] E. Soremekun, E. Pavese, N. Havrikov, L. Grunske, and A. Zeller. Inputs from hell. *IEEE Transactions on Software Engineering*, 48(4):1138–1153, 2020.

[35] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Efficient and accurate nearest neighbor and closest pair search in high-dimensional space. *ACM Transactions on Database Systems (TODS)*, 35(3):1–46, 2010.

[36] L. Van der Maaten and G. Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[37] G. Van Rossum, B. Warsaw, and N. Coghlan. Pep 8–style guide for python code. *Python. org*, 1565:28, 2001. Accessed: 2024-08-23.

[38] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley. Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 511–522, 2013.

[39] M. Zalewski. American fuzzy lop (afl). `https://lcamtuf.coredump.cx/afl/`, 2014. Accessed: 2024-10-01.

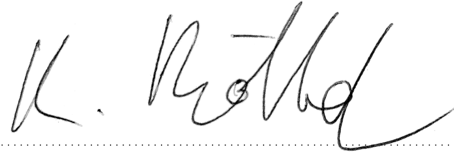
[40] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler. Testing configurations. In *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2023. Retrieved 2023-11-11 18:18:05+01:00.

[41] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler. Fuzzing with grammars. In *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2024. Retrieved 2024-06-30 18:31:28+02:00.

[42] Z. Zhang, G. Klees, E. Wang, M. Hicks, and S. Wei. Fuzzing configurations of program options. *ACM Transactions on Software Engineering and Methodology*, 32(2):1–21, 2023.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.



Berlin, den November 4, 2024 ....