API Knowledge Guided Test Generation for Machine Learning Libraries

ARUNKALEESHWARAN NARAYANAN

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF APPLIED SCIENCE

GRADUATE PROGRAM IN ELECTRICAL AND COMPUTER ENGINEERING

YORK UNIVERSITY
TORONTO, ONTARIO

July 2022

Abstract

This thesis proposes `MUTester` to generate test cases for APIs of machine learning libraries by leveraging the API constraints mined from the corresponding API documentation and the API usage patterns mined from code fragments in Stack Overflow (SO). First, we propose a set of 18 linguistic rules for mining API constraints from the API documents. Then, we use the frequent itemset mining technique to mine the API usage patterns from a large corpus of machine learning API related code fragments collected from SO. Finally, we use the above two types of API knowledge to guide the test generation of existing test generators, for machine learning libraries.

To evaluate the performance of `MUTester`, we first collected 2,889 APIs from five widely-used machine learning libraries (i.e., Scikit-learn, Pandas, Numpy, Scipy, and PyTorch),then for each API, we further extract their API knowledge, i.e., API constraints and API usage patterns. Given an API, `MUTester` combines its API knowledge with existing test generators (e.g., search-based test generator PyEvosuite and random test generator PyRandoop) to generate test cases to test the API. Results of our experiment show that `MUTester` can significantly improve the corresponding test generation methods. And the improvement in code coverage ranges from 18.0% to 41.9% on average.In addition, it also reduced 21% of invalid tests generated by the existing test generators.

Acknowledgments

I would like to express my deepest gratitude to my supervisor, Dr.Song Wang, for providing support and guidance throughout my graduate studies. He has provided valuable insights and feedback throughout my research. I could not have accomplished my objectives without his support.

Thanks to Dr.Maleknaz Nayebi, for the internship opportunity and mentoring, which greatly encouraged me to pursue research in software engineering.

My sincere thanks to Dr.Manar Jammal and Dr.Alvine Boaye Belle for their well wishes.

Thanks to all my friends and colleagues in our research lab who helped me throughout my studies. I would particularly like to thank Nima Shiri Harzevili for his assistance with my research.

I would like to thank my parents, sisters, and brothers-in-law for their support and encouragement.

Finally, I would like to thank God for the blessings, all my life.

Table of Contents

List of Tables

List of Figures

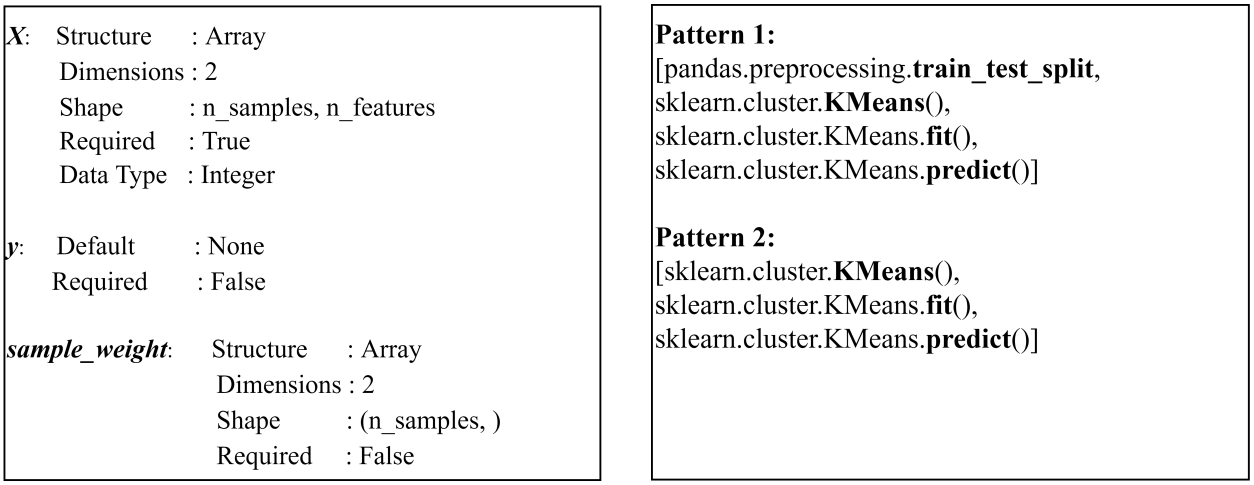
Chapter 1

Introduction

1.1 Motivation

To help generate test cases, many automated approaches that explore the input space and generate effective test cases for given APIs have been proposed in recent years, e.g., Evosuite [1], i.e., a typical search-based test generation technique and Randoop [2, 3], i.e., a feedback-directed random test generation approach. These test case generation tools have been widely examined on APIs of many traditional software systems, and results show that these tools are effective at generating unit test cases with high code coverage and can be used to help developers write high-quality unit test cases for APIs of traditional software systems. However, recent studies show that the performance of these test case generation tools on Machine Learning (ML) libraries are limited as many of the ML APIs expect inputs that follow ML-specific API knowledge, which cannot be provided by existing test generation tools [4, 5].

Specifically, existing test generators[1, 2] mainly target maximizing the code coverage under different strategies when synthesizing method call sequences or inputs for testing an API without considering the constraints of the API. However, generating valid and effective unit test cases for ML APIs requires two types of API knowledge, i.e., 1) the input constraints, i.e., the data structures of the input parameters such as arrays, lists, and tuples. For example, Figure 1.1a shows that `fit()` method of KMeans in Scikit-Learn library has three parameters, and each of them has different constraints, e.g., the first parameter `X` must be



(a) API constraints (b) API usage patterns

Figure 1.1: API Knowledge of method `KMeans.fit(X, y=None, sample_weight=None)` in `Scikit-Learn`

a two dimension array of numerical values, otherwise the function call to this method will crash; 2) API usage, i.e., the context of an API determined by the nature of the process of ML tasks. For example, Figure 1.1b shows two possible API usage patterns of the method `fit()`, in both the examples we can see that before calling method `predict(X)` to predict the closest cluster each sample in `X` belongs to, one needs to call method `fit()` to compute k-means clustering.

Most traditional test case generation tools often suffer from lack of the API knowledge, due to which they tend to generate test cases with invalid input values or incorrect method call sequences. For example, Figure 1.2a shows a test case developed by a search-based test case generation approach for `sklearn.cluster.KMeans`. The variables generated are tend to be in no accordance with the constraints of `sklearn.cluster.KMeans()`, e.g., the method requires its first parameter to be a numerical value and it is also a required parameter, but the generated test case passes a `None`, which makes the generated test case invalid. In addition, before calling `predict(X)`, one needs to call method `fit()` to compute k-means clustering, otherwise the generated API scenario is incorrect. This nature of lack of API knowledge leads existing test case generators produce many invalid test cases, since Python

```
import k_means as module_0
    def test_case():
        var0 = None
        var1 = [var0]
        var2 = 'm!"%SL2@D'
        var3 = None
        var4 = '6qRGnw"e='
        var5 = "k=ft\n<'?~GS\x0c|iD^"
        var6 = 'cYx\x0b< _RI5R'
        var7 = '\x0bD'
        var8 = var2, var5, var6, var7
        var9 = module0.KMeans(var3,n_init=var3,
              max_iter=var0,n_jobs=var4,algorithm=var8)
        var10 = module0.predict(var3)
        assert var9 is not None
```

(a) Test case generated by **PyEvosuite**.

```
import k_means as module_0
import _split as module_1

    def test_case():
        int_0 = 19
        var_0 = module_0.KMeans(int_0)
        float_0 = 1.6484
        int_2 = 69672
        float_1 = 83.14
        int_3 = 5568
        list_0 = [[float_0,int_2],[float_1,int_3]]
        var_1 = module_1.train_test_split(list_0)
        var_2 = var_0.fit(list_0)
        var_3 = var_0.predict(list_0)
        assert var_3 is not None
```

(b) Test case generated by `MUTester`.

Figure 1.2: Test cases generated by **PyEvosuite** and `MUTester`.

is a dynamic programming language, such invalid tests could pass without throwing any error, which potentially consumes the testing time yet without improving the coverage.

1.2 Objectives

To address the following limitations of existing test case generators,

- Generation of API calls with invalid inputs.
- Generation of incorrect API call sequences.

This thesis proposes `MUTester` to generate test cases for APIs of ML libraries by leveraging the API constraints mined from API documents and API usage patterns mined from code fragments in Stack Overflow. Figure 1.2b shows a test case generated by `MUTester`. We can see that the API is fed with valid inputs, generated with the guidance of the mined API constraints of `KMeans`. In addition, with the mined API usage patterns, the generated test case also has valid method calls to other APIs, which makes the test case more practical, thus achieving better code coverage.

We evaluate `MUTester`, in guiding two commonly-used automatic test case generation tools, i.e., the search-based test generation tool `Evosuite` and random test generation tool `Randoop`. We utilize the Python version of these tools (i.e., *PyEvosuite* and *PyRandoop*) implemented by the authors of Evosuite in [6]. To evaluate the performance of `MUTester`, we first collect 2,889 APIs from five widely-used ML libraries (i.e., Scikit-learn [7], Pandas [8], Numpy [9], Scipy [10], and PyTorch [11]) and for each API, we further extract its API knowledge, i.e., API constraints and API usage patterns. Given an API, `MUTester` combines its API knowledge and the existing test case generation method (e.g., search-based test generation or random test generation) to generate test cases for the API. Results of our experiment on the five ML libraries show that `MUTester` can significantly improve existing test case generation methods, i.e., an average of 15% More code coverage and an average of 21% Less invalid tests. This thesis makes the following contributions:

- We proposes `MUTester`(Figure 1.3) to generate test cases for APIs of ML libraries by leveraging the API knowledge mined from API documents and corresponding code fragments in Stack Overflow.
- We develop a set of 18 Linguistic rules for mining the API constraints from the API documentation, and we also leverage frequent itemset mining technique to mine API usage patterns from code fragments in Stack Overflow.

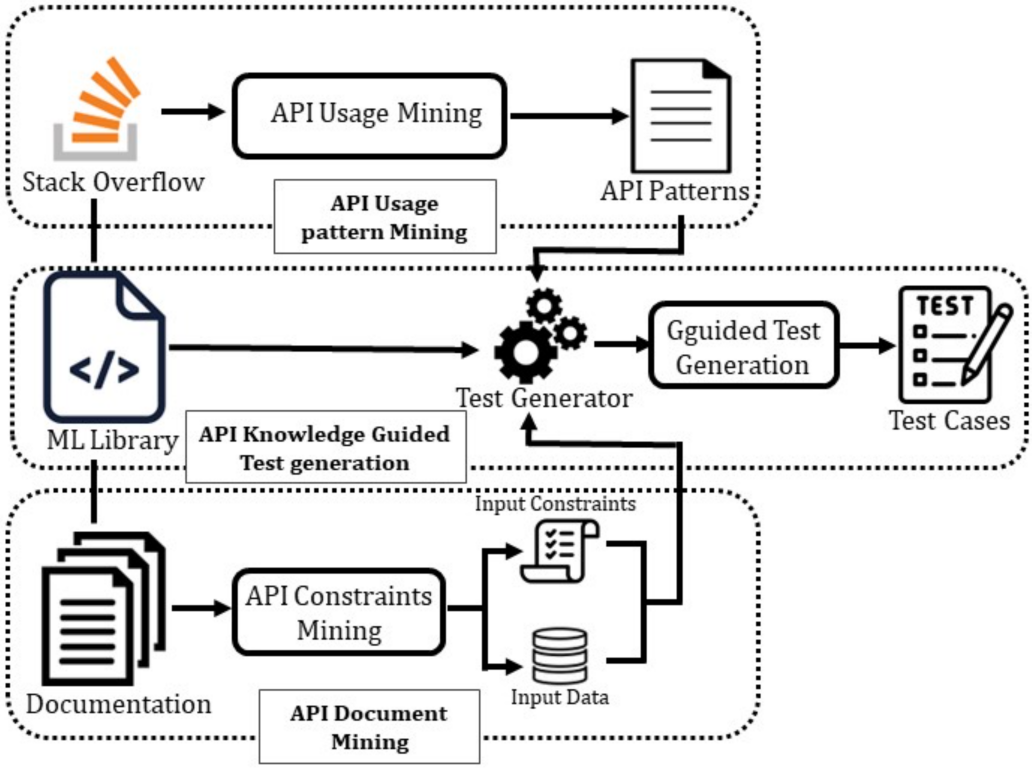


Figure 1.3: Overview of `MUTester`

- Our evaluation on five ML libraries shows the effectiveness of `MUTester` in improving code coverage and reducing invalid test cases.
- We release the source code of `MUTester` and the dataset of our experiments to help other researchers replicate and extend our study[1].

[1]https://doi.org/10.5281/zenodo.6525669

.

1.3 Outline

The thesis is structured into 7 main chapters as follows:

In Chapter 2, we present and explain the background of our research.

Chapter 3, 4, and 5 describes the methodology of our approach and related works for different components of the research.

Chapter 6 shows the experimental Setup and evaluation results for the research questions.

Finally, in Chapter 7, a summary of the significant results, conclusions, threats to validity and future work.

Chapter 2

Background

In this Chapter, we will give an abstract knowledge about important concepts and tools used throughout this thesis.

2.1 Data Mining

Data mining is the process involved in extraction of actionable knowledge from various sources of data available in the real world [12]. It can enable visualisation, analysis and exploration of large-scale datasets and hidden patterns in it. This broad research area of knowledge discovery in databases has received much attention and attraction in the past decade, due to wide availability of data sets, with higher volume and variety from almost all parts of the life [13].

Data mining can be carried out in databases using various techniques from Natural Language Processing [14] to Neural Networks [15, 16], and various algorithms from classification algorithms [17] to prediction algorithms [18]. Data mining is widely applied and effectively used in areas of education [19, 20], healthcare [21, 22], agriculture [23] and much more.

We use natural language processing, source code analysis, frequent itemset mining, and other data mining techniques in this research to obtain API knowledge from the data sources of API documents and stack overflow, which will, in turn, be used to generate guided test cases.

2.2 Unit Test cases

Unit test cases are executable code modules, which are developed manually or through automation [24] to test and validate individual units or components of a project such as certain class or method, in a whole software project [25]. Unit test cases can be helpful to test and validate the working of a piece of code in various conditions including borderline inputs, environments, and many other variables. This process can be helpful in finding and isolating bugs if there are any in the source code [26]. These test cases are more than often written by developers or testers to evaluate a specific piece of code, under test. Efficient unit tests can help developers to identify any bugs or faults in the code, which in turn paves way for a reliable software system.

In this thesis, as a major contribution, we propose an automated test generation tool `MUTester`, which can generate unit test cases for the source code modules of machine learning related APIs, by leveraging API knowledge.

2.3 Auto Test Generation

Automated test generators share a common process to generate test cases for a given API, i.e., synthesizing method call sequences that involves this API (i.e., synthesizing usage scenario) and generating inputs for parameters of each method in the call sequences [27] (i.e., synthesizing inputs). Note that, generating a strong oracle for a generated test requires a deep understanding of the logic of the test. Thus, most of the existing test generators ignore this step and check whether the generated test crashes or not. Although there exist many techniques for automatic test generation, this thesis focuses on two types of widely-adopted and scalable approaches for test case generation, i.e., random test generation and search-based test generation.

2.3.1 Random Test Generation

Random testing is a basic and scalable approach for test generation [28], which generates test cases by creating the invocation of functions with random inputs. Guided random testing is a refined approach that starts with random input data, then uses extra knowledge to guide input data generation. One typical example of random testing is feedback-directed random testing, i.e., Randoop [2, 3], which improves the random test generation by analysing the feedback collected from the previously generated test cases to avoid illegal input data. It starts by generating a random input data, then uses the feedback knowledge to develop input generation. Each of the generated test cases is executed immediately, to derive feedback and generate new ones.

2.3.2 Search-based Test Generation

Search-based test generation employs a genetic algorithm, that uses evolutionary operators like crossover, mutation, and selection to iteratively improve the candidate solutions for optimization of the fitness function. Fitness function in the search-based unit test generation often comprise code coverage of the generated test cases. Evosuite [1] is one of the typical search-based unit test generation tool. The working of EvoSuite across different types of software systems are researched and analysed by various empirical studies[29–33].

Note that, most of the existing test generation approaches, i.e., Evosuite [1] and Randoop [2, 3], are designed for object-oriented programming language such as Java and cannot be directly applied to the studied Python-based machine learning libraries.

In this work, we use the Python version search-based test generator (denoted as **PyEvosuite**) and random test generator (denoted as **PyRandoop**) implemented by the authors of Evosuite in [6] as our experiment subjects.

2.4 Machine Learning Libraries

Machine Learning, a subset of Artificial Intelligence, is used to enable a system to learn and solve the problem at hand by taking informed decisions based on the data from the past experience [34]. The process of machine learning, include the phases of data understanding, data preparation, modeling, evaluation phase, and deployment [35]. Each of these phases will involve the usage of various techniques and complex algorithms to achieve the desired goal. To facilitate the implementation of such techniques and algorithms by developers of different expertise, machine learning libraries are created and maintained in open source by organisations [36] and groups of developers [37].

Machine learning libraries provide implementation for various machine learning algorithms from linear regression [38] to convolutional neural network [39]. These implementations of machine learning libraries can facilitate developers to use complex algorithms, by invoking the respective API. Machine learning related libraries can also provide the implementation of algorithms, for different phases of the machine learning process: Pandas [40] and Numpy provide APIs for preprocessing and data visualization which are used in the phases of data understanding and data preparation [41].

2.5 API Knowledge

Modern software systems are often used and implemented in form of application programming interfaces(APIs) [42, 43]. These APIs help the developers use numerous functionalities of a software system by just invoking a single line of code function. To effectively use an API, understanding the nature and and working of the API is vital. Thayer et al. [44] in their detailed account of the theory of API knowledge, classified the types of knowledge into three types:

- **Domain concepts:** An abstract notion of what a particular API is designed to do, with its overall purpose. Since our test oracle concentrates on testing the proper functioning of an API code without crashing, rather than verifying the right implementation of its purpose, our tool is indifferent to the functions or correctness of the output, of the APIs under the test.

- **Execution concepts:** These facts or rules are constraints that should be followed while using an API. Implementing APIs without following such rules can result in incorrect functioning of the API or in some cases the program can even crash. This knowledge about the API includes parameter information, return type details, information about exception handling and others. The knowledge about the API constraints can be available in various sources like tutorials, documentations, code examples, code comments [45–52]. Our thesis proposes an NLP approach to extract API constraints required for test case generation from the API documents (Section 3).

- **API Usage Patterns:** These patterns are sequence or structure in which a certain API can be implemented along with other APIs, to achieve a specific goal. This knowledge is very useful in terms of test generation, as it can drive the tools to generate test cases with right API call sequence and check for their implementation. API related code fragments from several sources [53] like Stack overflow Question and answers, Github projects, example code in API documents can provide us with such knowledge about the API under consideration. In our proposed tool we extract API usage patterns from Stack overflow database and use them to generate test cases for machine learning related APIs.

For generating efficient test cases that can attain higher code coverage than that of the ones generated by existing test generation tools, `MUTester` mine and use API knowledge of API constraints (Execution concepts) and API usage patterns.

2.6 Frequent Item-set mining

Frequent Item-set mining is one of the important tasks in Data Science [54], used for finding regularities between items or variables. This type of data mining can greatly help in data analysis and decision-making process by uncovering hidden patterns in a transaction related dataset [55].

Let a transaction dataset with n distinct items is denoted by a non empty set $I = i1, i2, i3, ..., in$. Any transactional itemset with k items in it, is denoted by $X = (i1, i2, i3, \ldots, ik)$ with a specific transaction identifier (TID). The frequency of an itemset X can be described as the probability of k items in the set X occurring together in a transaction T, provided X is a subset of dataset I. The set X can be termed as a frequent item set, if its frequency support is greater than the user-specified threshold [56]. The support of item i in set I can be defined as:

$$Support(i) = \frac{Frequency(i)}{\#Transactions}$$

To enable API usage pattern guided call sequence generation in our proposed tool `MUTester`, we perform frequent item-set mining on API usage patterns mined from stack overflow code fragments.

2.7 Apriori Algorithm

Apriori is one of the simple and effective algorithm that can generate association rules by mining frequent itemsets [57]. It discovers the frequency of itemsets with k elements in each of its iterations by increasing the value of k by 1. It also eliminates itemsets that have support lesser than the user-defined threshold, for reducing the search space with each iteration. Apriori algorithm will involve searching the entire dataset, for calculating the frequency of each itemset, till no more frequent itemset is possible with the specified support threshold. In spite of reducing the search space with simple operations, the algorithm can provide good performance than other complex algorithms [58, 59] like FP-Growth [60], Equivalence

Class Transformation [61], Tree Projection algorithm [62], COFI algorithm [63] and TM algorithm [64].

Considering our requirements and the size of our dataset from stack overflow, we used the apriori algorithm for performing frequent item-set mining on API usage patterns.

Chapter 3

API Constraints Mining from API Documentation

3.1 Introduction

To help the end-users take full advantage of the APIs, the API documentation often provide publicly available information about the usage of each API [65] with the input formats and example code snippets, and etc. This information can be helpful for generating test cases with API input constraints and facilitating the generation of valid inputs.

However, manually analysing API documents and extracting constraints of APIs is time and effort consuming, For example, extracting parametric constraints of 1246 APIs in Scikit-Learn, will involve in inspecting 4980 Parametric lines, along with 346 Code examples. To tackle this challenge, we propose a semi-automated approach to collect following API constraints from the API documents:

- Data Structure (Array/List/Matrix/...)
- Data Type (Integer/String/...)
- Default Value of the parameter
- Shape (Dimension of the structure)
- Size (Range/Bounds of the data structure)

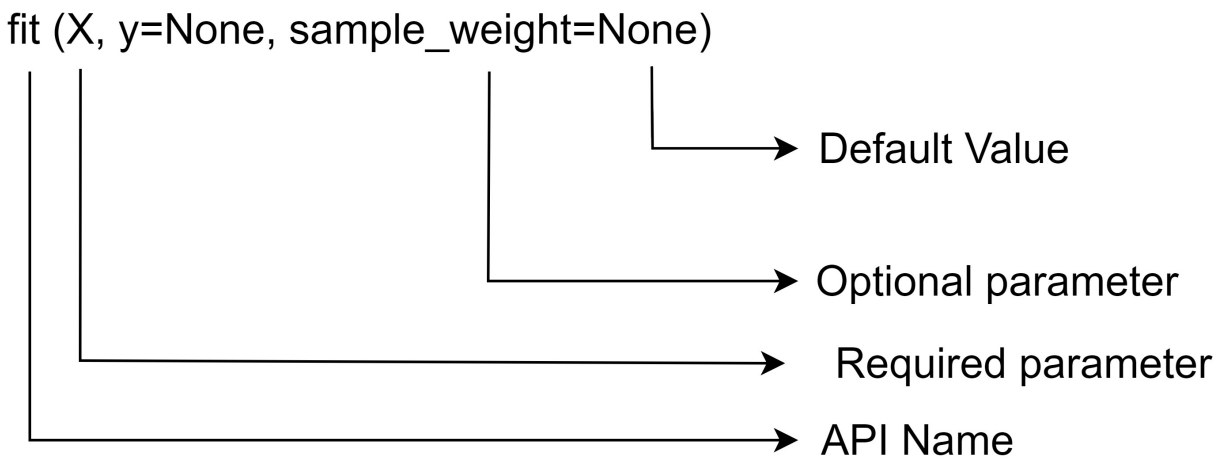


Figure 3.1: Header line of method `fit` from `sklearn.cluster.KMeans`

Parameters:

X : *{array-like, sparse matrix} of shape (n_samples, n_features)*
Training instances to cluster. It must be noted that the data will be converted to C ordering, which will cause a memory copy if the given data is not C-contiguous. If a sparse matrix is passed, a copy will be made if it's not in CSR format.

y : *Ignored*
Not used, present here for API consistency by convention.

sample_weight : *array-like of shape (n_samples,), default=None*
The weights for each observation in X. If None, all observations are assigned equal weight.

Figure 3.2: Parametric page of method `fit` from `sklearn.cluster.KMeans`

Given an API, we collect its API constraints from the following three sources.

3.1.1 Header line

The definition line for a class or a function, which shows the name of the class/function, and all the required and optional parameters. The header line also includes the default values for the optional parameters. For example, Figure 3.1 shows the header line of API `fit` from `sklearn.cluster.KMeans`.

```
from sklearn.cluster import KMeans
import numpy as np
X = np.array([[1, 2], [1, 4], [1, 0],
    [10, 2], [10, 4], [10, 0]])
kmeans = KMeans(n_clusters=2, random_state=0).fit(X)
kmeans.predict([[0, 0], [12, 3]])
```

Figure 3.3: Example code of `sklearn.cluster.KMeans`.

3.1.2 Parametric Line

The natural language sentences that describe the usage of a parameter with information about its data type, data range, fixed values, size and shape details in cases of arrays, matrix, list, and others. These sentences are normally given for each parameter of an API. For example, Figure 3.2 shows the parametric page of API `fit` from `sklearn.cluster.KMeans`, from which we can infer that the parameter `X` is an array.

3.1.3 Example code

The examples provided by a ML library that contain the usage of some APIs with concrete inputs. For each example code, we extract the concrete inputs and input types of data structures indicated in the parameter pages of the involved APIs. For example, Figure 3.3 shows the example code of `sklearn.cluster.KMeans`. To extract these data from the API documents, I designed a web scrapping tool using python based HTML parser [66].

3.2 Approach

3.2.1 Mining Header Line

As the format of the header line is in a structured format, we developed a regular expression-based string analyzer to mine and extract the following information from the header line,

- API Name: Name of the module
- Required Parameters: These parameters are the ones in the header line without default values. An API cannot be used without passing these required parameters.
- Optional Parameters: The parameters in header line with default values.
- Default Values: The values, an API assumes, if the parameter is not passed

For example, given the header line of method `fit` showed in Figure 3.1, our tool first identifies the API name is 'fit' and it has three parameters, i.e., 'X', 'y', and 'sample_weight'. In addition, 'X' is a required parameter as there is no default value for it in the header line. Both 'y' and 'sample_weight' are optional and the default values are 'None'.

It is also important to note that, as the header line reflects the source code, the information it contains are set to be final in case of mismatch in information between the header and parametric line.

3.2.2 Mining Parametric Page

Typically a Python API has one natural language described parametric sentence for each of its parameters in ML libraries to help users. We observe that, these natural language descriptions of parameters are of various formats and styles, which brings challenges for automatically extracting the parameter related information. Nevertheless, we also find that they would share similar linguistic patterns when mentioning the data type, data structure, etc, which motivated us in designing linguistic rules for the automatic extraction.

Table 3.1: API constraints extracted for API `fit` from `sklearn.cluster.KMeans` (Opt indicates whether the parameter is optional).

Parameter	Structure	Data Type	Default Value	Size	Shape	Opt
X	Array	Integer	N/A	2D	(n,n)	T
y	Undefined	Undefined	None	N/A	N/A	F
sample_weight	Array	float	None	N/A	n	F

To understand and design linguistic rules from the parametric pages, we conducted a manual analysis on documents of 100 randomly selected APIs from each of the ML libraries, i.e., Numpy, Pandas,Sickit-Learn,Scipy and Pytorch. In total 500 APIs were analyzed.

For our analysis, we first extracted all the parametric pages from the API documents of the studied 500 APIs, then we applied a pre-processing to remove noise tokens, e.g., white spaces and special characters [67].

After that, We worked to summarize possible regular expression-based linguistic patterns for each parameter and derive an initial list of rules for mining constraints for the parameter. Then we discussed the categorization results and merged similar ones, as a result of this process, 18 Rules are finalized to be used for mining parametric lines. These parametric rules are listed out with their examples in Table 3.2.

For example, rule No.1 in Table 3.2, a common pattern that appears in the parametric pages of the studied ML API documents, can be used to extract two types of API constraints from the matched example "int default=0", i.e., the data type and the default value of the parameter, "int" and "0" respectively.

3.2.3 Mining Example Code

To mine the example code, we build a heuristic-based static analyser to extract all the concrete input values of involved ML APIs from the example code, this help us to precisely infer properties of the API's input constraints, e.g., the concrete shape of the input data structure and the type of elements required in the input data structure. Note that not all the APIs have an example code snippet maintained by ML libraries, for these APIs that do

Table 3.2: Linguistic rules for mining constraints from parametric pages of APIs.

No	Constraint Type	Linguistic Rules	Examples
1	Data type, Default Value	<D_type> default=<value>	int default=0
2	Data type	<D_type> or <D_type>	int or float
3	Structure, Shape, Default Value	<Structure> of shape <(shape)>, default=<value>	array-like of shape (n_samples,n_features), default=None
4	Structure, Shape, Default Value	<Structure_Enum> of shape <(shape)>, default=<value>	{array-like, sparse matrix} of shape (n_samples, n_features), default=None
5	Structure, Shape	<Structure> of shape <(shape)> or <(shape)>	array-like of shape (n_samples,n_features) or (n_samples,)
6	Structure, Data Type	<D_type>/<structure>	params: dict
7	Structure	{Structure_Enum}	{list, tuple, set}
8	Values, Default Value	{Values_Enum} default=<value>	{'text', 'diagram'}, default=None
9	Values	{Values_Enum}	{'text', 'diagram'}
10	Size, Structure	<Size>length of {Structure_Enum}	2-length sequence (tuple, list, ...)
11	Dimension, Structure	<Dimension> d <structure>	2d Array
12	Deafult Value, Values	<value>(def), <value>, ... or <value>	'backward' (default), 'forward', or 'nearest'
13	Structure, Data Type	<Structure> of <D_type>	tuple of ints
14	Values	<value> or <value>	None or "sequence"
15	Data Type, Optionality	<D_type>, optional	(int,optional)
16	Structure, Optionality	<Structure>, optional	(array-like,optional)
17	Data Type, Values	<D_type> or <value>	int or "all"
18	Data Type, Values, Default Value	<D_type> or <value>default=<value>	int or "all", default=10

not have corresponding example code snippets, we reuse the proprieties of inputs from other APIs that share the same parameter pages. Given the example code showed in Figure 3.3, from the parametric page, we only know that the parameter X is an array, yet from this example code, we can further infer that it should be a 2-D integer array, which can facilitate inputs synthesizing with precise data information during test generation.

Table 3.1 shows the constraints mined for method `fit` of `sklearn. cluster.KMeans`, in which we can find the constraints for each of its three parameters. For example, parameter `X` is a required parameter and requires a 2D integer array. Once all the input constraints are mined, they will be used to generate test case for ML APIs.

3.3 Related Work

API documents and Stack Overflow are two important sources for API knowledge mining [68]. There are many existing studies that analyze API usage scenarios and extract useful patterns from API documents for specific tasks, e.g., API usage mining [69], API misuse detection [70], API constraints mining for fuzzing testing [71, 72], cloud API testing[73].

Maalej et al. [43] composed a study on knowledge organization in the reference documentation of APIs, in the platforms of Java SDK 6 and .NET 4.0. They discovered patterns of knowledge in API documentation using grounded methods and independent empirical validation conducted seventeen trained developers. This is more of an manual approach to present the taxonomy and knowledge patterns in API reference documentation. Li et al. [74] used NLP methods to discover API caveat sentences from API documentation and map them to entities in the API knowledge graph. To evaluate the approach, the authors build an API caveat knowledge graph on top of API documentation from Android applications. Maalej and Robillard [43] analyzed the subtle knowledge in API reference documentation to understand its structure and nature by mining many APIs in two well-known programming languages namely JAVA and C#. The authors used 17 Developers that are fully trained and used the created taxonomy to rate 5,574 Documents that are sampled randomly to evaluate the inherent knowledge that exists in them. The obtained results provide a set of patterns in API documentation such that this knowledge can be used to assist practitioners in asses the content of their API documentation. Zhong et al. [75] proposed a tool Doc2Spec, that can extract resource specifications from Javadocs using Natural Language processing techniques. These specifications were used to find bugs in open-source software that uses the APIs in their code. Zhai et al. [76] developed a text analysing engine using the Natural Language processing concepts of Parts-Of-Speech tagging and a tree transformer to extract information about APIs in JavaDoc. They used this data mining process to develop models for Java API functions. By this technique, they generated models for 326 Functions in 14 Java-based classes. This technique was developed towards mining API constraints only for Java-based

API functions through JavDocs. Dekel and Herbsleb [77] devised an Eclipse IDE plugin, for highlighting the sentences with API specifications from their respective API documents. It does not involve an automated extraction of API constraints, but needs human intervention in figuring out them, from the highlighted sentences. Pandita et al. [78] developed an approach to generate code contracts from the API documentation and domain dictionaries. They use POS tagging, semantic patterns, and other text analysis techniques to mine information needed for generating code contracts. Xie et al. [71] leveraged sequential pattern mining to generate rules for extracting deep learning specific constraints from API documents and uses these constraints to guide the fuzzing testing of deep learning frameworks. Subramanian et al. [79] proposed a technique to link API documentation to source code snippets in Q&A forum posts. They proposed a technique namely Baker which is capable of matching traditional API documentation with up-to-date source code examples with 97% Accuracy.

The main difference between the previous works and our study is two-fold, 1) we mine API knowledge from two different sources, i.e., API documents and Stack Overflow while they mine API constrains only from API documents. 2) we leverage the mined API knowledge to improve test case generation on python based Machine Learning related libraries.

Chapter 4

API Usage Patterns Mining from SO

4.1 Introduction

The common usage patterns of APIs can provide important clues for achieving a programming task and facilitating generating valid test cases [80, 81]. Most of the existing test case generation techniques [1, 28] randomly generate API call sequences to synthesize test cases, yet machine learning APIs are often used with specific order for different machine learning tasks. For example, given the two method `fit()` and `predict(X)` of`sklearn.cluster.KMeans` in Scikit-Learn library, before calling `predict(X)` method to predict the closest cluster each sample in `X` belongs to, one needs to call method `fit()` to compute k-means clustering. Such API usage patterns can be used to help test case generation with valid API call sequences and improve the coverage of the generated tests.

To generate valid API call sequences for ML libraries, following existing studies [82, 83], we use a frequent itemset mining based technique to mine API usage patterns from Stack Overflow code examples, which consists of two steps, i.e., API usage collection (Section 4.2.1) and API pattern mining (Section 4.2.2).

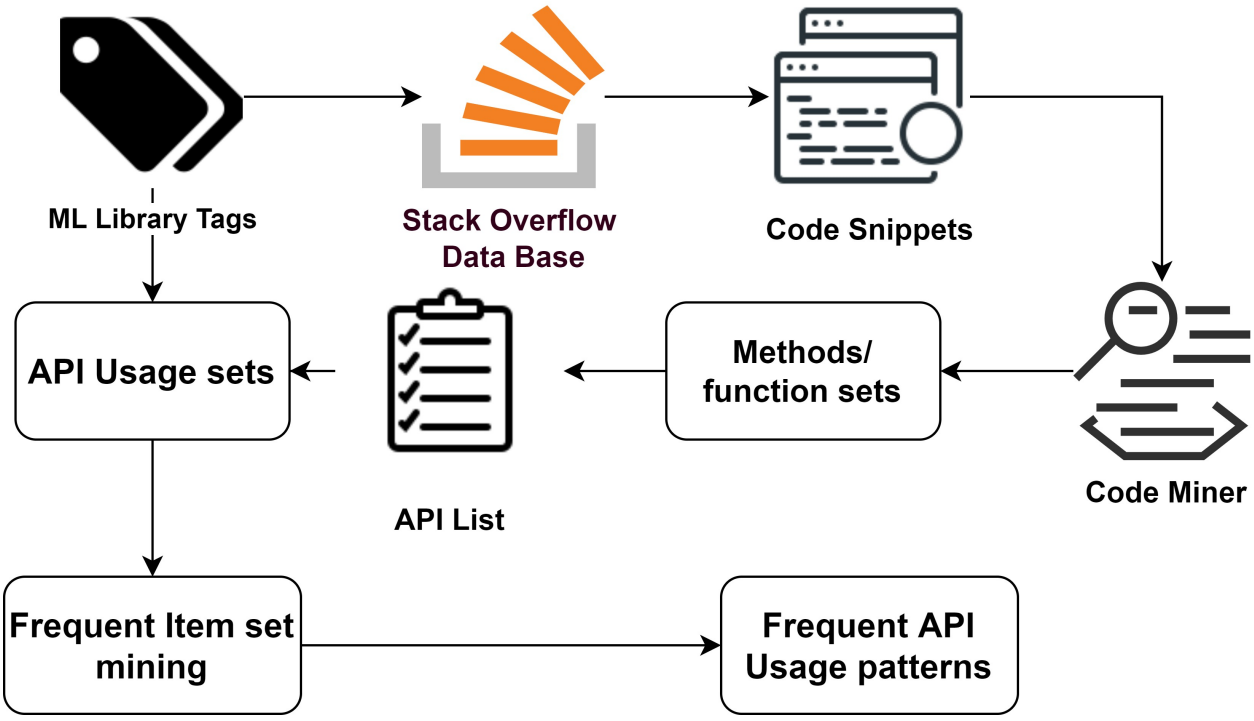


Figure 4.1: Overview of API Usage Pattern Mining from Stack Overflow

4.2 Approach

4.2.1 API Usage Collection

We follow existing work [83] to extract programming questions and their accepted answers from Stack Overflow using the data explorer provided by Stack Exchange [84]. We select questions with the name of the five ML library as Stack Overflow question tags to retrieve questions and only questions having accepted answers remain. We develop a heuristic-based APIs extraction method from the code fragments of accepted answers. To obtain the APIs used in all the code fragments of accepted answers, we designed static code analyser which can analyse a given code snippet and return a set of function call statements from the code snippets. These functions may include both APIs and user defined function. To eliminate user defined functions, we analysed Whether each function in the usage sets are APIs of ML libraries or not, by comparing the function names with the API names we extracted from the documentation of the ML related libraries scikit-learn, Pandas, Numpy, Scipy and Pytorch. Note that, If a question contains multiple APIs, we concatenate all APIs into a list of APIs.

We only select answers that contain at least two APIs for mining possible API usage patterns. As a result, we collected a total of 89,969 Question-API pairs, which involve 1,638 Unique APIs from the five studied ML libraries. After this, using the above said methodology we obtain the API usage dataset, which contains a set of API call sequences with each element being a list of APIs from the same code snippet. An abstract visualization of the API usage pattern mining from stack overflow is presented in the Figure 4.1

4.2.2 Mining API Usage Patterns

With the collected ML API usage dataset, we further mine the API usage patterns reflecting the co-occurrence relations of the APIs. We utilize the association rule mining technique on the API call sequences collected from the answers of ML programming tasks. Specifically, we use the Apriori Algorithm [57] to create a set of API association rules (i.e., API usage patterns) from the ML API usage dataset. We utilize the API call sequence from the answer of each SO post as input to the Apriori algorithm for rule mining. Apriori generates a pattern hypothesis by randomly selecting two or more APIs as the base itemset A and then randomly selecting one or more APIs as the extension itemset B. It further calculates the confidence and support of the pattern hypothesis $\{A \Rightarrow B\}$ and compares it to the confidence and support threshold specified by the user. Apriori accepts an association rule if the *confidence* and *support* of the rule are higher than the threshold values specified [57]. Specifically, *support* indicates the frequency of an API association rule with respect to the entire dataset. *Confidence* indicates the percentage of one or more extended API(s) (e.g., item set B) found to be true given a base rule (e.g., item set A).

In our experiment, we set *support* and *confidence* to 2 (e.g., the minimum support value in Apriori) and 50% to potentially recall more API usage patterns. After that we rank the patterns of each API based on their confidence values. These patterns will be further used in `MUTester` to help generate valid API call sequences. Note that, one can use small *confidence* values, i.e., smaller than 50%, to initialize the pattern generation process, while in our study, we find that a small *confidence* value does not change test cases generated by

`MUTester` given a reasonable time budge, i.e., 5 minutes for each API (as the top patterns are the same), while running Apriori with a small *confidence* value requires much more extra time. After this step, we output a list of API usage patterns descending ranked by the confidence values for guiding the test generation. List of such usage patterns mined for API *sklearn.linear_model.SGDClassifier* is given in table 4.1 as an example.

Table 4.1: Frequently used API usage patterns for `sklearn.linear_model.SGDClassifier`.

No	**API Usage Pattern**	**Confidence**
1	{SGDClassifier, fit}	34.28%
2	{SGDClassifier, fit, predict_proba}	30.00%
3	{SGDClassifier, partial_fit}	17.24%
4	{SGDClassifier, predict_proba}	13.79%

4.3 Related Work

The automated mining of crowd-sourced knowledge from SO has generated considerable attention in recent years [68, 83, 85, 86]. Uddin et al. [83] mined API documentation from stack overflow using automated techniques to analyse API usage scenarios. Huanget et al. [86] mined and used code fragments from stack overflow to recommend APIs to developer. They developed their recommendation system by analysing the probability of using one API with other.

Zhang et al. [69] developed an API usage mining framework equipped with a tool called MAPO (Mining API usage Pattern from Open-source repositories) for automatic mining of API usage patterns. Based on programs' requests, MAPO recommends the mined API usage patterns and their associated code snippets. The experimental results indicate that MAPO is effective at assisting programmers in programming tasks. Another interesting work is conducted by Nielebock et al. [70] mined API usage patterns to detect misuses in JAVA API used in open-source projects. Zhong et al. [87] conducted an empirical study to address a set of questions in API usage patterns. For instance, what are the common formats that define API usage scenarios and how to use a different types of APIs and etc. Uddin et al. [83] mined API documentation from stack overflow using automated techniques to analyse API usage scenarios. To this end, they proposed a novel framework in which it automatically links source code snippets in a forum post to an API in the textual contents of the same forum post. The framework subsequently provides a summary of the natural language text, in the forum post. Kavaler et al. [88] simultaneously analyzed data from both SO and Android marketplace to understand developer's intent, what programmers want to know and why. The motivation behind their study was, that developers have hard time understanding what they want from Q&A forums, which further motivated the authors to characterise the obstacles the developers have when using Q&A forums. Treude et al. [68] proposed an approach to augment API documentation with API knowledge mined from stack

overflow. Huang et al. [86] mined the API usage from SO to help recommend APIs for new queries posted in SO.

Different from these studies, in this work we perform frequent item-set mining on API usage patterns extracted from stack overflow, to enable our test generation tool `MUTester` to generate valid API call sequence in the test cases.

Chapter 5

API Knowledge Guided Test Generation

5.1 Introduction

After we extract the two types of API knowledge, i.e., API constraints (Section 3.2) and API usage patterns (Section 4.1), `MUTester` further leverages them to guide the test generation of existing approaches for ML libraries. Given a module from a ML library, existing test generators share a common workflow to generate tests which contains two major steps, i.e., method call sequence synthesizing, input synthesizing for these invoked methods. In this thesis, we use the mined API knowledge to guide these two steps with a target of generating more useful and valid tests given the same resource budget. Specifically, we leverage API usage patterns to narrow the search space for call sequence generation (see Section 5.2.1) and use API constraints to help solve the type information for input generation (see Section 5.2.2).

5.2 Approach

5.2.1 API Usage Patterns Guided Call Sequence Generation

Given a module from a ML library, existing test generators (i.e., **PyEvosuite** and **PyRandoop**) first parse the module and extract information about available methods in the module and its imports. The method call sequence construction often starts from creating the constructor of the module, during the execution, all the involved objects, methods, data,

statements, etc., will be stored for further use. After that under a specific criterion (e.g., **PyEvosuite** targets at maximizing code coverage when choosing the next methods), different methods will be added into the call sequence. Such process end when stopping conditions are triggered (e.g., time limitation, code coverage satisfied, maximum statements executed).

Current test generators do not consider the real-world API usage scenarios when constructing the method call sequence, thus can generate tests with incorrect API usages as shown in Figure 1.2a, i.e., before calling `predict(X)`, one needs to call method `fit()`, when using a K-Means algorithm provided by `sklearn.cluster.KMeans`. To solve this issue, `MUTester` leverages API usage patterns to guide the method call construction. Specifically, for a under expanding method call sequence M, `MUTester` will first iterate the available candidate method set and check whether a pattern can be matched based on M, if not, move to the longest sequential sub-sequence methods of M to check the usage patterns for the remaining methods and recursively. If multiple candidate methods can be matched, `MUTester` selects the one that has larger confidence and appends it to M. If no candidate method can be matched, `MUTester` uses the criterion of the adopted test generator to select the next method to be appended to M. Details are shown in Algorithm 1.

5.2.2 API Constraints Guided Input Generation

When a method is determined to be added into the method call sequence, a test generator needs to generate inputs for the method. Current generators such as **PyEvosuite** and **PyRandoop** mainly leverage the objects, data, and variables collected from last execution to initialize the parameters of the method and randomly guess the type information and inputs for parameters that are not available from the last executions, while ML APIs often require more complex and library specific structures like numpy.array, tuple, Date& time, tensor, etc., which often cannot be obtained from existing executions.

`MUTester` solves this issue by leveraging the API constraints mined from API documents. Specifically, given a parameter p from a method m, `MUTester` first obtains its basic type information regarding the data structure, the shape/size of data structure, data type, default

Algorithm 1 API usage guided call sequence construction

Require:
Call sequence under expanding M
The available candidate method set C
API pattern set P
Ensure:
Next method to be appended to M
initialize a priority method set $M_{priority}$
initialize a backup method set M_{backup}
for each candidate method c in C **do**
  while M.size() >1 **do**
    if $\{M >= c\}$ is in P **then**
      put c into $M_{priority}$
      BREAK
    end if
    $M = M.subList(1;)$
  end while
  if c is not in $M_{priority}$ **then**
    put c into M_{backup}
  end if
end for
sort $M_{priority}$ by confidence of patterns in P
sort M_{backup} by the criterion in test generator
if $M_{priority}$ is not empty **then**
  return top element in $M_{priority}$
else
  return top element in M_{backup}
end if

value, etc., mined from m's header line, parameter page, and example as illustrated in Section 3.2. `MUTester` then searches the last executions to see whether there exists matched inputs, if cannot find matches, `MUTester` further randomly generates inputs by following the constraints. Note that, as shown in Section 3.2.3, some of the concrete data type information of a specific data structure used in a method can only be inferred if the method appears in an example code, however not all APIs have a corresponding code example, following the existing test generators (e.g., **PyEvosuite** and **PyRandoop**), we randomly guess a primitive data type for them.

5.3 Related Work

To help developers generate tests, many automatic approaches have been proposed [1, 2, 89–92]. JCrasher [90] is one of the first random based test generator, which creates Java-based method call sequences that throw certain exceptions.

Similarly, Ecalt [89] and Randoop [2] employed the idea of random search to generate tests that can expose more faults. The major limitation of random testing is that it has no guidance about testing which results in low code coverage. Lagouvardos Et al. [93] developed a static analysis tool Pythia, which can track the tensor shape mismatch, in API implementation and application programs. This method of tracking the shape of tensors was able to find bugs related to shape mismatch in APIs of deep learning libraries. But this approach is only limited to checking the shape constraint of the API parameters. Wang et al. [81] analyzed the effectiveness of unit test generation techniques on machine learning libraries via conducting an empirical study on five well-known and actively used machine learning libraries using two famous automatic unit text case generation tools including Evosuite and Randoop. To mine constraints from API XML specifications and use them for test generation, Wang et al.[73] designed a structural and conditional analysis-based API Analyzer. Using the mined constraints, they also generated test cases to deploy in the cloud environment using the combinatorial data generation [94], heuristic graph search [95] and optimization algorithms. Their implementation is limited to cloud-based APIs and cloud environment. Machine learning libraries have their own constraints for API usage, which require test case generation techniques to be adaptable. Existing fuzzers are not able to deal with machine learning constraints. Hence, Xie et al. [96] proposed a novel document guided fuzzing technique namely D2C for API functions of DL libraries. To augment random testing, numerous approaches have been proposed including search-based algorithms and symbolic execution [1, 97, 98]. eToc [99] first employed genetic algorithms to test primitive data types and strings by generating test data. Along this line, Fraser et al. [1] proposed EvoSuite, which uses a genetic algorithm for automatic test case generation. The objective

function is maximizing code coverage based on a set of pre-defined criteria, e.g. covering all branches or statements, using evolutionary algorithms.

Kang et al. [100] presented an improved algorithm of grey-box based fuzzer ADFL, which is guided by code coverage. It was able to attain higher branch coverage than its predecessor AFL [101]. This fuzzer tool was checked on deeply nested programs, to evaluate its performance in attaining maximum branch code coverage. DocTer [72], a black box fuzzing-based test generation tool is developed to generate tests for Deep learning libraries. This tool can only generate test cases with random inputs based on fuzzing and does not implement the traditional test generation algorithms of Randoop and Evosuite. Bohme et al. [102] designed a directed grey-box fuzzer DGF as improvement to existing fuzzing-based test generators, by efficiently targeting the program location. It was able to outperform the traditional fuzzers, in projects like LibXML2. 17 [103], where security is critical. Its performance on ML libraries is not experimented with to be analyzed. OSS-Fuzz [104] and libFuzzer [105], are fuzzing based test generators that can be used to generate test cases for opensource APIs [106], but they suffer from the limitation of lack of API knowledge. Due to this they require manual input of API constraints to generate test cases.

Symbolic execution based test generation [107–110] is another approach to extending random testing where a program is executed abstractly in such a way that all possible combinations of inputs to a program is covered by considering execution path in a source code. However, symbolic execution-based approaches often suffer from a lack of scalability. The performance of test case generation on general software systems has been already addressed by existing experimental studies [81].

In this Thesis, we propose the first automatic unit tests generation for machine learning libraries, which uses API knowledge mined from API documents and stack overflow to generate more valid and effective test cases.

Chapter 6

Results and Analysis

6.1 Introduction

In order to summarize the performance of different components of our proposed test generation tool and the tool as its whole itself against existing baselines, we formulated following research questions and answer them in this chapter accordingly, with analysis of appropriate data.

- **RQ1: How effective are existing test generation techniques on ML Libraries?**
- **RQ2: How accurate is the mined API knowledge for ML libraries?**
- **RQ3: To what extent can the mined API knowledge improve existing test generation for ML libraries?**
- **RQ4: How effective are manually developed unit test cases of ML Libraries?**
- **RQ5: Can test cases generated using API Knowledge reduce invalid tests**

In this section we will discuss the results and conclusion arrived for each of the research question in detail.

Table 6.1: Experiment ML Libraries used in this paper.

ML Library	Description	#APIs	#Parameters	#SO posts	#Examples
Sickit-Learn	A library for classical machine learning algorithms.	1,210	4,980	8,735	346
Pandas	A library for data manipulation and analysis.	393	1,421	19,851	108
Numpy	A library for multi-dimensional matrix operations.	673	1,574	48,959	476
Scipy	A library for fundamental algorithms.	328	1,258	8,263	310
PyTorch	A deep learning library for computer vision and NLP.	285	833	4,161	174

6.1.1 Experiment Data

In this thesis, we use API documentation and modules from five commonly used ML-related libraries, i.e., Scikit-learn, Pandas, Numpy, Scipy, and PyTorch throughout our analysis. These ML libraries cover the current industrial machine learning practice and represent the critical aspects of machine learning developments. For instance, PyTorch provides high-level APIs to hide the low-level details of implementing deep learning applications. Scikit-Learn and Scipy are machine learning libraries with hundreds of APIs to build various machine learning models. Pandas and Numpy are two famous data analysis and visualization tools focusing on working with arrays, data frames, and support APIs. For each of the ML libraries, we extracted constraints of its APIs which have at least one parameter. Table 6.1 shows the details of each studied ML library regarding the number of APIs, the total number of parameters of all the APIs, the number of SO posts, and the number of code examples in the API documents of the ML library.

6.1.2 Evaluation Measure

Following existing studies [1, 30, 81], we utilize code coverage to evaluate the effectiveness of `MUTester` in test case generation. Code coverage of test cases determines the percentage of the code under the test has been executed and tested. Code coverage can be at different levels, e.g., branch level, statement level, and method level. In this work, following the existing test generators, i.e., **PyEvosuite** and **PyRandoop** [6], we measure the branch

level code coverage with `coverage.py`[1]. We run both test generators and `MUTester` on a 2.90GHz i7-10700F desktop with 16GB of memory.

6.2 RQ1: Effectiveness of Existing Test Generators on ML Libraries

6.2.1 Approach

To answer this RQ, we directly run the existing test generators, i.e., **PyEvosuite** and **PyRandoop**, on the five studied ML libraries. To generate tests for each module from the ML libraries, we used the default configurations and set the time limit to 5 minutes as suggested by [6, 81]. With the generated test cases, we further run `coverage.py` to collect the branch coverage on each module of each ML library. Note that, as both **PyEvosuite** and **PyRandoop** can generate flaky tests or tests with syntax errors, we have removed all these tests following [6, 81], i.e., we first run each test case and tests that throw *SyntaxError* will be filtered out, we then execute each non-syntax test five times, and removed flaky tests from the executions.This process was repeated until all remaining tests passed five times.

6.2.2 Result

Table 6.2: Performance of **PyRandoop** and **PyEvosuite** on the studied ML libraries.

ML Library	**PyRandoop**		**PyEvosuite**	
	#Test	**Coverage**	**#Test**	**Coverage**
Scikit Learn	1,573	43.52%	1,536	49.28%
Pandas	681	53.49%	713	57.73%
Numpy	992	36.50%	1,081	39.62%
Scipy	576	35.70%	519	41.83%
Pytorch	492	25.71%	558	27.80%
Average	862.8	37.87%	881.4	43.52%

[1]https://coverage.readthedocs.io/en/6.3.2/#

Table 6.2 shows the results of code coverage analysis on test cases generated by **PyEvosuite** and **PyRandoop** on the modules across the five ML Libraries. For each ML library, the table shows the number of generated tests and average code coverage achieved by the test generator. The number of generated tests varies dramatically for different ML libraries, this is mainly caused by the different API set sizes in these ML libraries. The average code coverage of an ML library is calculated by calculating the arithmetic mean of coverage achieved by test cases generated for each API of that library. As we can see from the table, overall the code coverage on these five ML libraries ranges from 25.71% (Pytorch) to 57.73% (Pandas) and **PyEvosuite** achieves higher code coverage on each ML library than that of **PyRandoop**. On average, **PyEvosuite** achieves the coverage of 43.52% on the five ML libraries while the average coverage of **PyRandoop** is 37.87%. We also conduct the Wilcoxon signed-rank test ($p < 0.05$) to compare the performance of **PyEvosuite** and **PyRandoop**. Results suggest that **PyEvosuite** achieves significantly better performance than **PyRandoop** on these ML libraries regarding code coverage.

Table 6.3: Improvement in code coverage for each ML library by **PyEvosuite**, than that of **PyRandoop**.

ML Library	**Coverage Improvement by PyEvosuite**
Scikit Learn	13.23%
Pandas	7.92%
Numpy	8.54%
Scipy	17.19%
Pytorch	8.12%

Table 6.3 further shows that in each ML library, **PyEvosuite** attains higher code coverage than of **PyRandoop**, this improvement in code coverage ranges from 8.12% (Pytorch) to 17.19% (Scipy). In spite of producing less number of test cases for the libraries of Scikit Learn and Scipy than of **PyRandoop**, **PyEvosuite** gives a clear improvement in code coverage. This is consistent with the evaluation of general Python projects [6] as **PyEvosuite** treats

test generation as an optimisation problem and applies evolutionary search algorithms to maximize code coverage.

The generated test cases by **PyRandoop** and **PyEvosuite** on the five studied ML libraries can only achieve less than half (37.87% and 43.52% respectively) of the code coverage.

6.3 RQ2: Accuracy of Mined API Knowledge

6.3.1 Approach

Table 6.4: Statistics of API usage patterns and API constraints extracted for each ML library.

ML Library	#API Usage Patterns	#API Constraints				
		Structure	Datatype	Default	Shape	Size
Scikit Learn	341	713	3,490	3,167	649	504
Pandas	473	437	1,092	1,279	285	213
Numpy	1,859	840	698	1,162	492	276
Scipy	297	371	861	1,008	260	235
Pytorch	172	317	371	782	214	206
Overall	3,142	2,398	6,512	7,398	1,900	1,434

To answer this question, we first follow our API knowledge mining approaches proposed in (Section 3.2) and Section 4.1 to mine API constraints (i.e., structure, data type, default values, and shape/size of the structure) and API usage patterns from API documents and SO posts respectively for all the APIs in the studied five machine learning libraries. The detailed statistics of mined constraints are shown in Table 6.4. Overall, we extract more than 2.3K, 6.5k, 7.3k, 1.9k, and 1.4k constraints for data structure, data type, default values, the shape of data structure, and size of data structure respectively. The number of API usage patterns mined for different ML libraries varies(Figure 6.1), e.g., our approach mines 1.8k API usage patterns for Numpy while only 172 patterns are mined for Pytorch. This is because Numpy is a very popular library and there are more questions in SO than in any other four libraries, thus it has more code fragments and more API usage patterns are inferred.

To evaluate the accuracy of these API constraints mined, we randomly selected 200 parameters from each of the five libraries, making up to a total of 1,000 API parameters. We then manually check the header lines, parametric pages, and code examples to collect the ground-truth API constraints. After that two of our authors will work together to manually

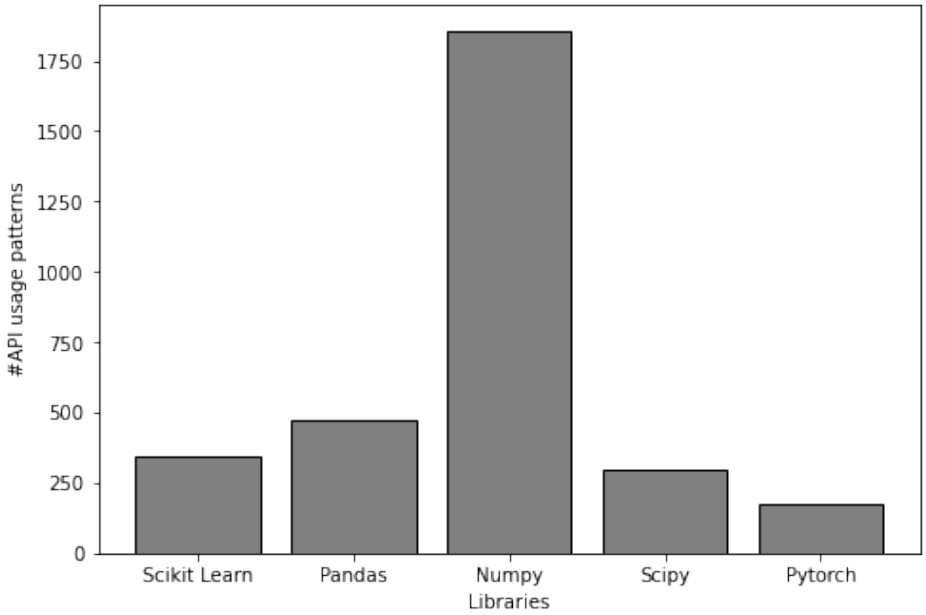


Figure 6.1: API Usage Patterns Mined for each ML Libraries

verify the mined constraints against the ground truth in the API documents. Note that as we mine the API usage patterns from the code fragments of accepted answers of SO questions, we assume all the API usage patterns are correct.

6.3.2 Result

Table 6.5: Accuracy of mined constraints.

Library	**Accuracy of Constraints**				
	Structure	**Data Type**	**Shape**	**Size**	**Default**
Scikit-Learn	91.72%	95.28%	100%	100%	90.75%
Pandas	89.57%	93.70%	100%	100%	94.50%
Numpy	86.19%	88.30%	84.32%	84.32%	86.38%
Scipy	80.40%	81.05%	79.36%	100%	82.64%
Pytorch	90.33%	92.75%	87.14%	87.14%	97.27%

Table 6.5 shows the accuracy of mined constrains from each ML library. Overall, the accuracy of constraint extraction is higher that 85% across different libraries. While the performance on different library vary, e.g., the accuracy of 'Shape' related constraint extracted

from Scikit-learn is 100% while on Scipy, the accuracy is only 79.36%, this could be caused by the diversity of API document formats and also the quality of documents provided by different libraries, for example most of Scipy's parametric pages do not mention the default values even if the corresponding parameters have default value settings.

The constraints mined from the studied ML libraries have high accuracy, which shows the usefulness of our constraint extraction approaches to facilitate the follow-up guidance for test generation.

6.4 RQ3: Improvement of Test Generation with API Knowledge

6.4.1 Approach

To evaluate the performance of `MUTester` in test generation for machine learning related APIs, we first collect all the modules from each of the five ML libraries. Then we mined the API constraints and API usage patterns, according to Section 3 and Section 4. Based on this, we input the API modules and the corresponding API knowledge into our developed tool `MUTester` for performing API knowledge-guided test generation. For each module, we run `MUTester` for five minutes as we did in RQ1.

With the generated test cases, we further run `coverage.py` to measure the branch-level code coverage for the API modules. As we did in RQ1, we also remove the generated flaky tests or tests with syntax errors, i.e., we discard tests that throw *SyntaxError* and then we execute each non-syntax test five times to remove potential flaky tests.

6.4.2 Result

Table 6.6: Code coverage results for test cases generated by `MUTester` and the improvement to **PyEvosuite** and **PyRandoop** .

ML Library	**MUTESTER**					
	Randoop			**Evosuite**		
	#Tests	**Coverage**	**Improvement**	**#Tests**	**Coverage**	**Improvement**
Scikit Learn	1609	49.27%	13.21%	1588	62.59%	27.0%
Pandas	622	61.95%	15.87%	697	68.17%	18.0%
Numpy	1157	42.08%	15.28%	1143	56.23%	41.9%
Scipy	681	42.14%	18.03%	633	51.19%	22.5%
Pytorch	538	28.43%	10.57%	529	32.97%	18.6%

Table 6.6 shows the number of generated tests and the corresponding code coverage attained by `MUTester` on the modules across the five ML Libraries. In this experiment we show the performance of `MUTester` based on both **PyEvosuite** and **PyRandoop**. We have

observed much improvement with `MUTester` when built on **Evosuite** than that of the **Randoop**. Like RQ1, this again shows the superior performance of **Evosuite** than compared to the performance of **Randoop**.

Overall, `MUTester` achieves better code coverage on each ML library compared to the existing approaches of **PyEvosuite** and **PyRandoop**.The improvement in code coverage ranges from 18.0% (Pandas) to 41.9% (Numpy) in the implementation of **Evosuite**. With the implementation of **Randoop**, again `MUTester` shows improvement in code coverage ranging from 10.57%(Pytorch) to 15.87%(Pandas). This result specifies that irrespective of the test generation algorithm with which it is implemented, `MUTester` can improve the code coverage with the help of mined API knowledge.

In addition, for four out of the five ML libraries, `MUTester` can cover more than half of the code, showing its superiority in performance over the existing approaches. Our Wilcoxon signed-rank test ($p < 0.05$) also suggests that `MUTester` can significantly outperform both **PyEvosuite** and **PyRandoop**.

> `MUTester` can significantly improve performance of test generation for ML libraries. The improvement against existing test generators can be up to 41.9% (Numpy) in code coverage.

6.5 RQ4: Performance of Manually developed test cases and MUTester

6.5.1 Approach

To understand the performance of unit test cases developed by developers of the machine learning libraries, combined with the test cases generated by `MUTester`, we first collect all the unit test cases available in the five ML libraries and run `coverage.py` on each of the test cases to calculate the code coverage achieved by the unit test cases. Then in order to analyse the improvement in code coverage when the these test cases are combined with the one generated by `MUTester`, we combine the code coverage files obtained from the Section 6.4 and the coverage files of unit test cases, to calculate the total branch coverage attained by the test cases developed by `MUTester` and the manually developed test cases.

6.5.2 Results

Table 6.7: Coverage results for Unitest cases with `MUTester`.

ML Library	**Unit test cases**	**Mutester(Evosuite)+Unit test cases**	
	Coverage	**Coverage**	**Improvement**
Scikit Learn	67.25%	69.97%	4.04%
Pandas	69.18%	72.69%	5.07%
Numpy	52.61%	60.28%	14.57%
Scipy	48.84%	57.04%	16.78%
Pytorch	58.47%	59.16%	1.18%

Table 6.7 projects the coverage results of Unit test cases in the machine learning repositories and the themselves combined with the test cases generated by **Evosuite** implementation by our tool `MUTester` as it outperforms the implementation of **Randoop**. It is also important to note that not all API modules in a repository is covered by the available unit test case and one unit test can also be developed to test more than one API. overall the combined code coverage out performs the code coverage by both the unit test cases and the test cases

by `MUTester`. The improvement ranges from 1.18% to 16.78%. Note that the performance of the unite test cases in the library depends on several factors including the developer skills and maintenance of the software. So the coverage of these unit test case significantly differ from each library, whereas the performance of the test cases developed by `MUTester` depends on the availability of API knowledge

The combined coverage of test cases in ML libraries and test cases generated by `MUTester` outperforms all the baselines, indicating that the generated test cases along with Unit test cases can yield great result in terms of code coverage.

6.6 RQ5: To What Extent Can `MUTester` Help Reduce Invalid Tests?

6.6.1 Approach

We have noticed that both **PyEvosuite** and **PyRandoop** generate many invalid tests. These invalid tests are neither flaky nor have syntax errors, yet they take incorrect inputs which violate the parameter constraints and would not work as expected. For example, if an API with parameter p only requires an integer variable, but the test case passes a string variable, then we call it an invalid test. As Python is a dynamic programming language, such tests could pass without throwing any errors, yet could not produce the anticipated results. Since our proposed `MUTester` is designed as integrating the API knowledge with existing test generators, it can potentially help reduce the invalid tests.

To explore to what extent, `MUTester` can help reduce these invalid tests generated by existing generators, we first randomly collect 100 tests generated by **PyEvosuite** on these five ML libraries as **PyEvosuite** performs better than **PyRandoop**, then we manually check whether the generated tests are invalid. After that, for the verified invalid tests, we regenerate them with `MUTester` and manually check whether `MUTester` can correct their parameter issues.

6.6.2 Result

Table 6.8 shows the number of invalid tests generated by **PyEvosuite** and the number of invalid tests that `MUTester` can help avoid. Overall, **PyEvosuite** generates around 60% invalid tests across the five ML libraries. We can also observe that the number of invalid tests on Numpy and Scipy is larger than on others. The possible reason is that both of them provide APIs mainly related to fundamental algorithms, which require more complex inputs. We can also see that with the help of `MUTester`, in total 62 out of the 292 Invalid tests can be removed. `MUTester` fails to correct the left invalid tests mainly because of a lack of detailed data type information which is mined from code examples, as for most of

Table 6.8: Invalid tests generated by **PyEvosuite** and can be removed by `MUTester`.

ML Library	**Invalid Tests**	**Removed by `MUTester`**
Scikit Learn	43	9
Pandas	49	9
Numpy	73	14
Scipy	68	16
Pytorch	59	14
Overall	292	62

the APIs there is no example code provided in the ML libraries. In future work, we plan to mine the code examples from other sources, e.g., Stack Overflow and GitHub, to retrieve more useful knowledge for the parameter settings.

`MUTester` can help remove 21% (62 out of 292) Invalid tests generated by **PyEvosuite** on the five ML libraries, which suggests the practice value of `MUTester`.

Chapter 7

Conclusion

7.1 Summary

The main goal of this thesis was to develop an efficient python based automated test case generation tool for Machine Learning Libraries which leverages API Knowledge mined from API documents and Stack Overflow code fragments. As a result, we performed detailed empirical analysis on existing test generation tools and test cases already present in the Machine Learning Libraries. We used the results from our empirical study as our baseline to compare and analyse the working of our test case generation tool `MUTester`

First, in Chapter 3, we proposed the first component of our test case generation tool which involves in Mining parametric sentences, header lines and example code snippets from the API documents of the API under test. We introduced and explained the process we use to extract information from the API documents that are used for test generation. In this chapter, we published 18 Parametric mining rules, which can be used to extract parametric information of APIs from API documents.

Secondly, in Chapter 4, we proposed the next component of our tool. This chapter explains the mining of Frequently usage patterns from the code fragments in the answers of Stack Overflow, an online Q&A forum.We give a detailed account on various techniques and measures we follow to extract the frequent usage patterns.

Thirdly, in Chapter 5, we established how we use the data and information acquired from previous chapters, to generate test cases that are guided with the knowledge of API

documents and API Usage patterns. In this chapter we also published the algorithm, we used to integrate the usage patterns to the API call sequence generation in generating test cases.

Finally, in Chapter 6, we formulated and answered six research questions, with our extensive analysis and comparison of the metrics derived from the test cases generated by `MUTester` and several other baselines. We also explained the nature and setup for the experiments carried out to analyse the results .

7.2 Threats to Validity

7.2.1 External Validity

Factors or threats affecting the generalizability of the `MUTester` and the research, to other environments is discussed in the following

- As `MUTester` is developed to generate test cases for Python based APIs, at its current capacity, it cannot generate tests for modules written in any other programming language other than Python. But the approach of API knowledge guided test generation can be extended to other languages in future works.

- Even though the approach of mining constraints from header lines and example code in the API documents is generalizable across various API reference documents, the 18 Linguistic rules proposed to mine parametric lines may not be applied to other machine learning related libraries with different API documenting styles.

- Our approach of API knowledge guided test generation for machine learning related APIs have significant improvement in code coverage than traditional methods, but the efficiency of this approach on traditional software systems with minimal or zero documentation, is subject to evaluation.

- Guided test generation in `MUTester` is developed with the algorithms of Randoop and Evosuite, It may not generate testcase along with other test generation algorithms like MOSA [111].

7.2.2 Internal Validity

Threats to internal validity refers to the factors that can affect the assumptions and decisions made in overall approach of our thesis.

- In this thesis, we use Evosuite and Randoop algorithms implemented by the authors of Evosuite in [6] as our experiment subjects, so the correctness of their implementations can affect the performance of `MUTester`.

- The correctness of constraints acquired from mining API documents is based on the assumption that, the API documents are properly written and maintained in accordance with the source code of the API implementation. If there is some wrong information about the APIs in the documents, our tool does not have the ability to identify them.

- It cannot be guaranteed that all the code snippets mined from stack overflow correctly represent the usage of the APIs, as they are constructed by independent developers.

- In experimentation, the time budget of five minutes allocated to the test generation is based on the original implementation of `PyRandoop` and `PyEvosuite`. Changes in this time budget can increase or decrease the code coverage attained on the source code of the APIs.

7.2.3 Construct Validity

Threats to construct validity examines the shortcoming in measuring values used to evaluate an approach.

- We use code coverage to assess the efficiency of generated test cases based on the assumption that code coverage and test effectiveness has a correlation with each other. There are other measures for this purpose such as mutation score, which we plan to use to examine the effectiveness of tests in the future.

- The accuracy of constraints mining is calculated based on the ground truth developed by us. So the ground truth can be prone to human errors.

7.3 Future Work

The tools and techniques developed and the conclusions claimed by this thesis can be further supported and extended in future works. I have listed some of the future works that can be performed with the contributions of this thesis.

- **Experimentation:** The performance of the developed test generation tool, `MUTester` can be further experimented with other machine learning libraries like Tensorflow [112], Keras [113] and CNTK [36], to support and verify our claims.

- **API constraint mining:** `MUTester` can be extended to generate test cases for a wide range of API-related libraries, by improving the natural language processing technique used to mine constraints, from API documentation.

- **API usage patterns:** In our work, we mined the usage patterns only from the stack overflow database. But in future works, to mine usage patterns, several other data sources like machine learning related projects in Github [114] and Kaggel [115] platforms can be considered to widen the dataset.

- **Programming languages:** Our proposed tool `MUTester` was developed to perform only with Python-based modules. But the approach we proposed to leverage the mined API knowledge for test generation, can be extended to other programming languages also.

- **Test generation algorithms:** `MUTester` can be extended to perform with other test generation algorithms like Many-Objective Sorting algorithm(MOSA) [111] and Many Independent Objective(MIO) algorithm [116].

7.4 Final Conclusion

This thesis proposes `MUTester` to generate test cases for APIs of ML libraries by leveraging the API constraints mined from the corresponding API documentations and the API usage patterns mined from code fragments in Stack Overflow (SO). Given an API, `MUTester` combines its API knowledge with existing test generators to generate test cases to test the respective API. Results of our experiment on five widely-used machine learning libraries (i.e., Scikit-learn, Pandas, Numpy, Scipy, and PyTorch) show that `MUTester` can significantly improve the corresponding test generation methods and the improvement in code coverage is 18.0% to 41.9% on average. In addition, it can also help reduce around 21% of invalid tests generated by the existing test generators.

Bibliography

[1] G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software," in *FSE'11*, 2011, pp. 416–419.

[2] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for java," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, 2007, pp. 815–816.

[3] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *29th International Conference on Software Engineering (ICSE'07)*, 2007, pp. 75–84.

[4] S. Wang, N. Shrestha, A. K. Subburaman, J. Wang, M. Wei, and N. Nagappan, "Automatic unit test generation for machine learning libraries: How far are we?" In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1548–1560.

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

[6] S. Lukasczyk, F. Kroiß, and G. Fraser, "Automated unit test generation for python," in *International Symposium on Search Based Software Engineering*, Springer, 2020, pp. 9–24.

[7] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux, "API design for machine learning software: Experiences from the scikit-learn project," in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 2013, pp. 108–122.

[8] T. pandas development team, *Pandas-dev/pandas: Pandas*, version latest, Feb. 2020.

[9] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020.

[10] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey,
bibinitperiodI. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[11] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in.

[12] B. Lakshmi and G. Raghunandhan, "A conceptual overview of data mining," in *2011 National Conference on Innovations in Emerging Technology*, 2011, pp. 27–32.

[13] M.-S. Chen, J. Han, and P. Yu, "Data mining: An overview from a database perspective," *IEEE Transactions on Knowledge and Data Engineering*, vol. 8, no. 6, pp. 866–883, 1996.

[14] J. Yi, T. Nasukawa, R. Bunescu, and W. Niblack, "Sentiment analyzer: Extracting sentiments about a given topic using natural language processing techniques," in *Third IEEE international conference on data mining*, IEEE, 2003, pp. 427–434.

[15] M. W. Craven and J. W. Shavlik, "Using neural networks for data mining," *Future generation computer systems*, vol. 13, no. 2-3, pp. 211–229, 1997.

[16] H. Lu, R. Setiono, and H. Liu, "Effective data mining using neural networks," *IEEE transactions on knowledge and data engineering*, vol. 8, no. 6, pp. 957–961, 1996.

[17] S. S. Nikam, "A comparative study of classification techniques in data mining algorithms," *Oriental Journal of Computer Science and Technology*, vol. 8, no. 1, pp. 13–19, 2015.

[18] P. Baldi, S. Brunak, Y. Chauvin, C. A. Andersen, and H. Nielsen, "Assessing the accuracy of prediction algorithms for classification: An overview," *Bioinformatics*, vol. 16, no. 5, pp. 412–424, 2000.

[19] M. M. Arcinas, G. S. Sajja, S. Asif, S. Gour, E. Okoronkwo, and M. Naved, "Role of data mining in education for improving students performance for social change," *Turkish Journal of Physiotherapy and Rehabilitation*, vol. 32, no. 3, pp. 6519–6526, 2021.

[20] M. Goyal and R. Vohra, "Applications of data mining in higher education," *International Journal of Computer Science Issues (IJCSI)*, vol. 9, no. 2, p. 113, 2012.

[21] N. Jothi, W. Husain, *et al.*, "Data mining in healthcare–a review," *Procedia computer science*, vol. 72, pp. 306–313, 2015.

[22] I. Yoo, P. Alafaireet, M. Marinov, K. Pena-Hernandez, R. Gopidi, J.-F. Chang, and L. Hua, "Data mining in healthcare and biomedicine: A survey of the literature," *Journal of medical systems*, vol. 36, no. 4, pp. 2431–2448, 2012.

[23] B Milovic, V Radojevic, *et al.*, "Application of data mining in agriculture.," *Bulgarian Journal of Agricultural Science*, vol. 21, no. 1, pp. 26–34, 2015.

[24] D. Winkler, R. Hametner, and S. Biffl, "Automation component aspects for efficient unit testing," in *2009 IEEE Conference on Emerging Technologies & Factory Automation*, IEEE, 2009, pp. 1–8.

[25] H. Zhu, P. A. Hall, and J. H. May, "Software unit test coverage and adequacy," *Acm computing surveys (csur)*, vol. 29, no. 4, pp. 366–427, 1997.

[26] P. Hamill, *Unit test frameworks: tools for high-quality software development.* " O'Reilly Media, Inc.", 2004.

[27] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2012.

[28] A. Arcuri, M. Z. Iqbal, and L. Briand, "Random testing: Theoretical results and practical implications," *TSE'11*, vol. 38, no. 2, pp. 258–277, 2011.

[29] G. Fraser and A. Arcuri, "A large-scale evaluation of automated unit test generation using evosuite," *TOSEM'14*, vol. 24, no. 2, pp. 1–42, 2014.

[30] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, "Does automated unit test generation really help software testers? a controlled empirical study," *TOSEM'15*, vol. 24, no. 4, pp. 1–49, 2015.

[31] R. S. Herlim, S. Hong, Y. Kim, and M. Kim, "Empirical study of effectiveness of evosuite on the sbst 2020 tool competition benchmark," in *International Symposium on Search Based Software Engineering*, Springer, 2021, pp. 121–135.

[32] T. Virgínio, L. A. Martins, L. R. Soares, R. Santana, H. Costa, and I. Machado, "An empirical study of automatically-generated tests from the perspective of test smells," in *Proceedings of the 34th Brazilian Symposium on Software Engineering*, 2020, pp. 92–96.

[33] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, "An industrial evaluation of unit test generation: Finding real faults in a financial application," in *ICSE-SEIP'17*, 2017, pp. 263–272.

[34] M. I. Jordan and T. M. Mitchell, "Machine learning: Trends, perspectives, and prospects," *Science*, vol. 349, no. 6245, pp. 255–260, 2015.

[35] G. Nguyen, S. Dlugolinsky, M. Bobák, V. Tran, Á. López García, I. Heredia, P. Malík, and L. Hluchỳ, "Machine learning and deep learning frameworks and libraries for large-scale data mining: A survey," *Artificial Intelligence Review*, vol. 52, no. 1, pp. 77–124, 2019.

[36] F. Seide and A. Agarwal, "Cntk: Microsoft's open-source deep-learning toolkit," in *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, 2016, pp. 2135–2135.

[37] R. Kohavi, D. Sommerfield, and J. Dougherty, "Data mining using a machine learning library in c++," *International Journal on Artificial Intelligence Tools*, vol. 6, no. 04, pp. 537–566, 1997.

[38] G. A. Seber and A. J. Lee, *Linear regression analysis*. John Wiley & Sons, 2012.

[39] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," *arXiv preprint arXiv:1511.08458*, 2015.

[40] I. Meniailov, K. Bazilevych, K. Fedulov, and S. Goranina, "Using the k-means method for diagnosing cancer stage using the pandas library," *development*, vol. 14, p. 15, 2019.

[41] S. Li and A. Kumar, "Morpheuspy: Factorized machine learning with numpy," Technical report, 2018. Available at https://adalabucsd. github. io/papers ..., Tech. Rep.

[42] C. R. De Souza and D. F. Redmiles, "On the roles of apis in the coordination of collaborative software development," *Computer Supported Cooperative Work (CSCW)*, vol. 18, no. 5, pp. 445–475, 2009.

[43] W. Maalej and M. P. Robillard, "Patterns of knowledge in api reference documentation," *IEEE Transactions on Software Engineering*, vol. 39, no. 9, pp. 1264–1282, 2013.

[44] K. Thayer, S. E. Chasins, and A. J. Ko, "A theory of robust api knowledge," *ACM Trans. Comput. Educ.*, vol. 21, no. 1, 2021.

[45] A. Head, C. Sadowski, E. Murphy-Hill, and A. Knight, "When not to comment: Questions and tradeoffs with api documentation for c++ projects," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018, pp. 643–653.

[46] W. G. Lutters and C. B. Seaman, "Revealing actual documentation usage in software maintenance through war stories," *Information and Software Technology*, vol. 49, no. 6, pp. 576–587, 2007.

[47] A. J. Ko, R. DeLine, and G. Venolia, "Information needs in collocated software development teams," in *29th International Conference on Software Engineering (ICSE'07)*, IEEE, 2007, pp. 344–353.

[48] C. Parnin and C. Treude, "Measuring api documentation on the web," in *Proceedings of the 2nd international workshop on Web 2.0 for software engineering*, 2011, pp. 25–30.

[49] E. Duala-Ekoko and M. P. Robillard, "The information gathering strategies of api learners," Technical report, TR-2010.6, School of Computer Science, McGill University, Tech. Rep., 2010.

[50] A. J. Ko, B. A. Myers, and H. H. Aung, "Six learning barriers in end-user programming systems," in *2004 IEEE Symposium on Visual Languages-Human Centric Computing*, IEEE, 2004, pp. 199–206.

[51] M. Meng, S. Steinhardt, and A. Schubert, "Application programming interface documentation: What do software developers want?" *Journal of Technical Writing and Communication*, vol. 48, no. 3, pp. 295–330, 2018.

[52] J. Nykaza, R. Messinger, F. Boehme, C. L. Norman, M. Mace, and M. Gordon, "What programmers really want: Results of a needs assessment for sdk documentation," in *Proceedings of the 20th annual international conference on Computer documentation*, 2002, pp. 133–141.

[53] J. Stylos and B. Myers, "Mica: A web-search tool for finding api components and examples," in *Visual Languages and Human-Centric Computing (VL/HCC'06)*, 2006, pp. 195–202.

[54] J. M. Luna, P. Fournier-Viger, and S. Ventura, "Frequent itemset mining: A 25 years review," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 9, no. 6, e1329, 2019.

[55] C. Borgelt, "Frequent item set mining," *Wiley interdisciplinary reviews: data mining and knowledge discovery*, vol. 2, no. 6, pp. 437–456, 2012.

[56] J. Pillai and O. Vyas, "Overview of itemset utility mining and its applications," *International Journal of Computer Applications*, vol. 5, no. 11, pp. 9–13, 2010.

[57] R. Agrawal, R. Srikant, *et al.*, "Fast algorithms for mining association rules," in *Proc. 20th int. conf. very large data bases, VLDB*, Citeseer, vol. 1215, 1994, pp. 487–499.

[58] C.-H. Chee, J. Jaafar, I. A. Aziz, M. H. Hasan, and W. Yeoh, "Algorithms for frequent itemset mining: A literature review," *Artificial Intelligence Review*, vol. 52, no. 4, pp. 2603–2621, 2019.

[59] A. Inokuchi, T. Washio, and H. Motoda, "An apriori-based algorithm for mining frequent substructures from graph data," in *European conference on principles of data mining and knowledge discovery*, Springer, 2000, pp. 13–23.

[60] J. Han and J. Pei, "Mining frequent patterns by pattern-growth: Methodology and implications," *ACM SIGKDD explorations newsletter*, vol. 2, no. 2, pp. 14–20, 2000.

[61] M. J. Zaki, "Scalable algorithms for association mining," *IEEE transactions on knowledge and data engineering*, vol. 12, no. 3, pp. 372–390, 2000.

[62] R. C. Agarwal, C. C. Aggarwal, and V. Prasad, "A tree projection algorithm for generation of frequent item sets," *Journal of parallel and Distributed Computing*, vol. 61, no. 3, pp. 350–371, 2001.

[63] M. El-Hajj and O. R. Zaiane, "Cofi-tree mining: A new approach to pattern growth with reduced candidacy generation," in *Workshop on frequent itemset mining implementations (FIMI'03) in conjunction with IEEE-ICDM*, 2003.

[64] M. Song and S. Rajasekaran, "A transaction mapping algorithm for frequent itemsets mining," *IEEE transactions on knowledge and data engineering*, vol. 18, no. 4, pp. 472–481, 2006.

[65] Y. Hashemi, M. Nayebi, and G. Antoniol, "Documentation of machine learning software," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020, pp. 666–667.

[66] L. Richardson, "Beautiful soup documentation," *April*, 2007.

[67] K. Chowdhary, "Natural language processing," *Fundamentals of artificial intelligence*, pp. 603–649, 2020.

[68] C. Treude and M. P. Robillard, "Augmenting api documentation with insights from stack overflow," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, IEEE, 2016, pp. 392–403.

[69] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "Mapo: Mining and recommending api usage patterns," in *European Conference on Object-Oriented Programming*, Springer, 2009, pp. 318–343.

[70] S. Nielebock, R. Heumüller, K. M. Schott, and F. Ortmeier, "Guided pattern mining for api misuse detection by change-based code analysis," *Automated Software Engineering*, vol. 28, no. 2, pp. 1–48, 2021.

[71] X. Xie, J. W. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen, "Testing and validating machine learning classifiers by metamorphic testing," *JSS'11*, vol. 84, no. 4, pp. 544–558, 2011.

[72] Y. Li, "Documentation-guided fuzzing for testing deep learning api functions," M.S. thesis, University of Waterloo, 2020.

[73] J. Wang, X. Bai, L. Li, Z. Ji, and H. Ma, "A model-based framework for cloud api testing," in *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2, 2017, pp. 60–65.

[74] H. Li, S. Li, J. Sun, Z. Xing, X. Peng, M. Liu, and X. Zhao, "Improving api caveats accessibility by mining api caveats knowledge graph," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2018, pp. 183–193.

[75] H. Zhong, L. Zhang, T. Xie, and H. Mei, "Inferring resource specifications from natural language api documentation," in *2009 IEEE/ACM International Conference on Automated Software Engineering*, 2009, pp. 307–318.

[76] J. Zhai, J. Huang, S. Ma, X. Zhang, L. Tan, J. Zhao, and F. Qin, "Automatic model generation from documentation for java api functions," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 380–391.

[77] U. Dekel and J. D. Herbsleb, "Improving api documentation usability with knowledge pushing," in *2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 320–330.

[78] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar, "Inferring method specifications from natural language api descriptions," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 815–825.

[79] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live api documentation," in *Proceedings of the 36th international conference on software engineering*, 2014, pp. 643–652.

[80] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t)," in *ASE'15*, 2015, pp. 201–211.

[81] S. Wang, N. Shrestha, A. K. Subburaman, J. Wang, M. Wei, and N. Nagappan, "Automatic unit test generation for machine learning libraries: How far are we?" In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, IEEE, 2021, pp. 1548–1560.

[82] T. Zhang, G. Upadhyaya, A. Reinhardt, H. Rajan, and M. Kim, "Are code examples on an online q&a forum reliable?: A study of api misuse on stack overflow," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, IEEE, 2018, pp. 886–896.

[83] G. Uddin, F. Khomh, and C. K. Roy, "Mining api usage scenarios from stack overflow," *Information and Software Technology*, vol. 122, p. 106 277, 2020.

[84] *Query stackoverflow - stack exchange data explorer*, Available at https://data.stackexchange.com/stackoverflow/query/new (2021/08/01), 2021.

[85] F. Calefato, F. Lanubile, M. C. Marasciulo, and N. Novielli, "Mining successful answers in stack overflow," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, IEEE, 2015, pp. 430–433.

[86] Q. Huang, X. Xia, Z. Xing, D. Lo, and X. Wang, "Api method recommendation without worrying about the task-api knowledge gap," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2018, pp. 293–304.

[87] H. Zhong and H. Mei, "An empirical study on api usages," *IEEE Transactions on Software Engineering*, vol. 45, no. 4, pp. 319–334, 2017.

[88] D. Kavaler, D. Posnett, C. Gibler, H. Chen, P. Devanbu, and V. Filkov, "Using and asking: Apis used in the android market and asked about in stackoverflow," in *International Conference on Social Informatics*, Springer, 2013, pp. 405–418.

[89] C. Pacheco and M. D. Ernst, "Eclat: Automatic generation and classification of test inputs," in *European Conference on Object-Oriented Programming*, Springer, 2005, pp. 504–527.

[90] C. Csallner and Y. Smaragdakis, "Jcrasher: An automatic robustness tester for java," *Software: Practice and Experience*, vol. 34, no. 11, pp. 1025–1050, 2004.

[91] S. L. K. Pond and S. V. Muse, "Hyphy: Hypothesis testing using phylogenies," in *Statistical methods in molecular evolution*, Springer, 2005, pp. 125–181.

[92] J. P. Shaffer, "Multiple hypothesis testing," *Annual review of psychology*, vol. 46, no. 1, pp. 561–584, 1995.

[93] S. Lagouvardos, J. Dolby, N. Grech, A. Antoniadis, and Y. Smaragdakis, "Static analysis of shape in tensorflow programs," in *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[94] K. Z. Zamli, N. A. M. Isa, M. F. J. Klaib, and S. N. Azizan, "A tool for automated test data generation (and execution) based on combinatorial approach," *International Journal of Software Engineering and Its Applications*, vol. 1, no. 1, pp. 19–35, 2007.

[95] C. Yang, S. Tian, and B. Long, "Application of heuristic graph search to test-point selection for analog fault dictionary techniques," *IEEE Transactions on Instrumentation and Measurement*, vol. 58, no. 7, pp. 2145–2158, 2008.

[96] D. Xie, Y. Li, M. Kim, H. V. Pham, L. Tan, X. Zhang, and M. Godfrey, "Leveraging documentation to test deep learning library functions," *arXiv preprint arXiv:2109.01002*, 2021.

[97] L. Baresi, P. L. Lanzi, and M. Miraz, "Testful: An evolutionary test approach for java," in *ICST'10*, 2010, pp. 185–194.

[98] A. Leitner, I. Ciupa, M. Oriol, B. Meyer, and A. Fiva, "Contract driven development= test driven development-writing test cases," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2007, pp. 425–434.

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

[100] C. Wang and S. Kang, "Adfl: An improved algorithm for american fuzzy lop in fuzz testing," in *International Conference on Cloud Computing and Security*, Springer, 2018, pp. 27–36.

[101] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "{afl++}: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[102] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2329–2344.

[103] D. Veillard, "The xml c parser and toolkit of gnome. libxml," *System Home page at http://xmlsoft. org*, 2003.

[104] K. Serebryany, *Oss-fuzz*.

[105] K. Serebryany, "Continuous fuzzing with libfuzzer and addresssanitizer," in *2016 IEEE Cybersecurity Development (SecDev)*, IEEE, 2016, pp. 157–157.

[106] Z. Y. Ding and C. Le Goues, "An empirical study of oss-fuzz bugs," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, IEEE, 2021, pp. 131–142.

[107] W. Visser, C. S. Păsăreanu, and S. Khurshid, "Test input generation with java pathfinder," in *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, 2004, pp. 97–107.

[108] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[109] N. Tillmann and J. d. Halleux, "Pex–white box test generation for. net," in *International conference on tests and proofs*, Springer, 2008, pp. 134–153.

[110] C. S. Păsăreanu and N. Rungta, "Symbolic pathfinder: Symbolic execution of java bytecode," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2010, pp. 179–180.

[111] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *TSS'17*, vol. 44, no. 2, pp. 122–158, 2017.

[112] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015.

[113] F. Chollet *et al.* (2015). "Keras," [Online]. Available: https://github.com/fchollet/keras.

[114] *Github*, https://github.com/.

[115] *Kaggle: Your machine learning and data science community*, https://www.kaggle.com/.

[116] A. Arcuri, "Many independent objective (mio) algorithm for test suite generation," in *International symposium on search based software engineering*, Springer, 2017, pp. 3–17.