Determine the Origin of Python Software Source Code using the Distinctiveness of Identifiers

by

Yiming Sun
B.Sc., University of Victoria, 2019

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Determine the Origin of Python Software Source Code using the Distinctiveness of Identifiers

by

Yiming Sun
B.Sc., University of Victoria, 2019

Supervisory Committee

Dr. Daniel M. German, Supervisor
(Department of Computer Science)

Dr. Alex Thomo, Department Member
(Department of Computer Science)

Dr. Katsuro Inoue, Outside Member
(Department of Software Engineering, Faculty of Science and Technology, Nanzan University)

Supervisory Committee

Dr. Daniel M. German, Supervisor
(Department of Computer Science)

Dr. Alex Thomo, Department Member
(Department of Computer Science)

Dr. Katsuro Inoue, Outside Member
(Department of Software Engineering, Faculty of Science and Technology, Nanzan University)

ABSTRACT

More than 75% of organizations rely on open source as the foundation of their applications. The prevalence of code reuse in modern software development processes has led to potential drawbacks for developers and companies on both security and legal aspects, particularly for the inability to identify the origin of a copied software artifact. Conventional methods, such as code clone detection approaches, might not address this problem effectively, mainly because they are too exhaustive to be practical. Hence, I propose a lightweight, scalable, and robust method to narrow down the origin of a Python source code entity to a few possible candidates within a reference corpus of the Python Packaging Index (PyPI) open-source ecosystem, using only a few extracted classes and function name identifiers. Then, more exhaustive methods become applicable to this small set of candidates to identify the exact origin.

Analyzing the PyPI ecosystem, I found that identifiers are very distinct. Among 11.2 M different identifiers found within PyPI's 244 K packages, 76% are only defined in one package, and 93% are in at most 3. Generally, randomly selecting 3 non-frequent identifiers from one or multiple files from an input package is enough to narrow down the origin to at most 3 packages 89% of the time. I evaluated the proposed method by mapping Debian Python packages to corresponding PyPI packages,

where only 3 identifiers are extracted from each Debian package and matched against PyPI packages. I used a popularity index to rank the returned candidate packages so that the top one is the most likely to be the origin. By empirical experiments, this method is effective at finding the correct origin of a package that is not directly collected from the corpus, with a recall of 87%.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to thank every friend, department member, and staff who fulfilled my university life with joy, knowledge, and art. The seven years at UVic and in Canada will be the most memorable fragment of my life. I would like to express my particular gratitude to:

My parents, Weilu Sun and Guiju Liu, for their unconditional love and understanding & supporting all my decisions.

My supervisor, Dr. Daniel German, for providing the opportunity of pursuing a master's degree, and for the mentoring, patience, and encouragement even when I was not good enough; from him, I learned to think and act as a researcher.

Dr. Stefano Zacchiroli, for sharing his knowledge and effort, which greatly helped my research.

My wife, Yuanyuan Wang, for her smiles and immense love.

DEDICATION

This thesis is dedicated to my parents, who gave me life, hope, and strength to pursue all my dreams.

Chapter 1

Introduction

The prevalence of code reuse is a well-known characteristic of modern software engineering, attracting significant attention in the research field and industries. As a typical pattern, software systems are developed by integrating internal custom code with external open-source software (OSS) artifacts that can be publicly and freely retrieved from various locations. Mockus (2007) [54] reported that, in the scope of several large open-source repositories, more than 50% of the source code files' names occurred in different projects. According to Securosis 2014 "Open Source Development and Application Security Survey" [2], at least 75% of organizations rely on open source as the foundation of their applications. A more recent 2020 report "Open Source Security and Risk Analysis" by Synopsys [74] indicated that 99% of the audited codebases in 2019 contained OSS artifacts, with 70% of the codebases being open-source themselves.

Code reuse may occur in various manners during development, depending on *how* and *why* the OSS artifacts are reused by developers. As one common manner, developers sometimes require cloning entire OSS code repositories into their software products that are being built, which is as known as "software vendoring" (i.e., the act of making one's copy of the third-party packages that their project is using [35]). Such practice often happens when the reused artifacts are libraries that the products rely on as dependencies. The products are packaged along with these libraries (embedded dependencies), then distributed to end users. The purpose behind it is to ensure library compatibility—the products will run in consistent environments that have been tested stable, which involves libraries with particular versions, regardless the users have them already installed or not on their platforms. Another potential reason for software vendoring is that some products may require too many libraries, making it

difficult to install the software successfully. For example, a Python project `kolibri`[1] is bundled with over 50 libraries in its source package.

Another typical manner of code reuse is copying only fractions of OSS artifacts, such as source code files, functions, or chunks of code. Similarly, code can be copied from public reference sources like programming Q&A sites where developers can look up code snippets. The purpose of this reuse is that programmers try to opportunistically accomplish coding tasks like implementing certain features while avoiding coding from scratch due to its expense in both time and effort. Haefliger et al. [75] found that developers reuse code to finish their development work more efficiently; sometimes, they lack the skills to implement certain functionality by themselves, or they want to deliver a "credible promise" with their work. Such a manner of using readily and freely accessible code artifacts is commonly believed to have benefits on code quality, coding efficiency, and software maintenance [21]. Furthermore, per Open Source ideology, developers tend to not only reuse code but also advertise their high-quality code for others to reuse on code-sharing sites such as GitHub Gist, Pastebin, and Codeshare [21].

1.1 Motivation

Code reuse benefits software development in various aspects, yet documenting the origins of the reused OSS artifacts is vital for stakeholders properly manage their software assets. In other words, developers, managers, QA engineers, and other stakeholders should be able to identify if a given source code *artifact* from their software product—that is, a library, file, function, chunk of code, etc.—was copied/adapted from an external OSS component, and if so, from where and how the artifact came to be what it currently is. The reason for such information is so important has been considered from two perspectives:

First, the technical and security perspective is being considered chiefly related to the practices of vendoring, since maintainers of software products would monitor the status of all the external libraries that have been bundled inside the distributed packages and keep them updated when newer versions that contain vulnerability fixes are released. Once the origin information of a cloned library is missing, that is, the library's name and/or version can no longer be identified, such library might

[1] `https://pypi.org/project/kolibri/0.14.7/`, last accessed 2022-10-13

no longer be updated, then the potential vulnerabilities might harm a product and lead to serious security issues. 88% of applications audited in a study contained open source dependencies that had no updates in the last two years [74]. Moreover, a "bi-directional" security concern may arise from the side of libraries that had been used. For example, in the mentioned project `kolibri`, assuming that one of the cloned libraries, namely `requests`, was detected containing a vulnerability in a specific version, the developers of `requests` may wish to know what other open source projects such as `kolibri` have cloned that release as an integrated dependency, for the sake of being able to determine the vulnerability's impact and warn whom in charge of such projects. According to an executive order signed by the president of the USA on improving the nation's cyber security [1], software supply chain security is a crucial part of it. It requires the ability to quickly determine whether software systems were developed securely, for which software artifact origin determination might play an essential role.

Second, from a legal point of view, OSS artifacts are not always "free-to-use", since under certain circumstances reusing might cause license (either open source or commercial) violations. Before distributing a software product to clients, its distributor is responsible for ensuring that the licenses associated with all the involved OSS artifacts are compatible with each other and consistent with the client software license [68]. Short of that, a software product might confront legal and financial risks that are embarrassing to address. Checking the licenses of the reused artifacts also requires basic origin information of the component. In addition, from the opposite side, OSS owners should somehow be able to enumerate existing software products that have reused their components, to detect prohibited usages.

Therefore, failing to identify reused OSS artifacts and their origin can lead to technical and legal issues. For this reason, stakeholders demand approaches to help them identify the origin of source code artifacts. The preliminary requirement of any of those approaches is to locate one or more OSS artifacts that, to a certain extent, are identical (or almost identical) to the artifact being investigated within a finite range of possible candidates. Although code matching can already be performed with mature code clone detection methods, finding matches in vast candidate scopes such as commonly known OSS repositories (e.g., GitHub) remains challenging. On the one hand, a significant portion of these methods is based on textual or Abstract Syntax Tree (AST) comparisons, which are too computationally expensive to be practical. On the other hand, other fast clone detection methods, such as cryptographic hash

file fingerprinting, have the disadvantage that when minor changes (even editing comments) are applied to the source file, the recomputed hash cannot match the original file; moreover, file hashing does not work for non-file granularity matching, such as finding the occurrences of a file fragment (i.e., code snippet) in a set of files.

In summary, a good code matching approach is still in demand for origin identifying, which should be *lightweight* to perform, *scalable* to various matching granularities and large candidate sets, and *robust* to minor changes.

1.2 The Problem of Software Origin Determination

The goal of this thesis is to develop a method for the efficient identification of the potential *origin* of a source code *artifact* given a reference *corpus*. Within the scope of this thesis, a software artifact is one or more source code files, and the corpus is a reference collection of software *products* (i.e., projects) from where those files might have been copied. Therefore, the problem this thesis solves is: given a *subject* (a file or set of files), identify the software product (within the corpus) from where these files might have been copied.

In practice, this method will not always find the exact origin of the subject. Instead, the method could return multiple software products equally likely to be the origin, and these products are called *candidates*; ideally, one for them is the subject's origin. A suitable origin determination method is expected to narrow down the search space by returning as minimum as a possible number of candidates, ideally one. When the candidate size becomes small, other more exhaustive (computationally expensive) approaches (e.g., clone detection, AST matching, or even human inspection) become adoptable to identify the actual origin within the candidates.

1.3 Proposed Solution

The proposed solution for the software origin determination problem is by taking advantage of the fact that developers tend to choose global identifiers that are unique to their software product. In the context of this thesis, global identifiers correspond to programming language entities with a global scope in the software system where they are defined (e.g., globally defined classes, functions, packages, etc.).

By programming conventions and industrial specifications, identifiers should be descriptive such that they convey semantic meanings to programmers, also identifiers are proposed to follow specific syntactic rules for different contexts (e.g., required by PEP 8 [3]: use `PascalCase` that also named `CapWords` for class names, `snake_case` for function names, `UPPER_SNAKE_CASE` for constants, etc.). Hence intuitively, identifiers should be unique so that within a collection of OSS artifacts, we might find extremely few artifacts that define a given identifier.

Therefore, I conjecture that most identifiers in a given corpus containing most software products in an ecosystem are unique. This property can be leveraged to identify the origin software product from which a file (or set of files) has been copied. For the proposed approach, I would focus on using two types of global identifiers: **class** and **function** (including class method) names.

I specifically consider the OSS ecosystem of Python software packages distributed by the Python Package Index (PyPI) [4] as the domain of my corpus, from which I seek empirical evidence to validate the effectiveness of the approach. PyPI is a comprehensive repository of Python projects and is the default source of Python's official package manager PIP. The first step of this thesis was to build a corpus from those packages accessible from PyPI by creating a mapping between global identifiers and the packages where they have been defined. By doing that, I have created a corpus of 244 K software packages based on a snapshot of PyPI as of August 2020.

With this corpus, the method to identify the origin of a subject (a file or set of files) is:

1. Randomly extract a small set of identifiers from the subject.
2. Return the candidate packages that define all these identifiers.
3. Rank the candidates according to the popularity index, SourceRank[2]), which is a software product evaluation metric based on both the quality and popularity indexes.

Ideally, only one package should be returned by this method. Otherwise, as mentioned before, we expect the number of packages returned to be small, so other methods (e.g., clone detection) can be used to properly identify the files' origin.

The approach of using identifiers to determine the origin has three merits:

[2]`https://docs.libraries.io/overview.html`, last accessed 2022-10-13

1. **Lightweight**, since the identifiers are small portions of information drawn from code, an identifier-indexed reference corpus is relatively compact in terms of storage usage compared to the original size of code artifacts. More importantly, it efficiently queries the identifiers from an artifact against such a corpus.

2. **Scalable**, as the approach is possible to be used for a corpus at the size of massive OSS artifacts, also the subject and origin can be software artifacts in various levels of granularities, such as an entire software project, a set of files, or one or more functions.

3. **Robust** to minor changes in code as long as the original identifiers have been preserved.

This work is divided into three parts, with one research question proposed and answered in each part. The first research question aims to validate the conjecture that is the most fundamental premise of using the approach:

RQ 1: How unique are the class and function names among open-source Python ecosystem?

In the context of this thesis, I let *distinctiveness* of an identifier be the number of products in which such identifier is defined with respect to a reference corpus. I answered this question by measuring the distinctiveness of all the identifiers in the corpus, then computing their distribution to provide an overall picture of the corpus, bringing insights into whether identifiers can effectively be used for our purpose. The results showed that identifiers are distinctive: about 76% of identifiers are defined in exactly one product (i.e., *uniquely* identify a product), and about 93% of the identifiers appear in at most 3 different products; furthermore, if a name is randomly chosen from a random product within the corpus, an identifier is unique with a probability of 32%.

In the next part of the thesis, I explored how efficient this approach is for finding the origin of a file (or set of files) using only a small number of identifiers. Hence, the second research question is:

RQ 2: How many identifiers are needed to *narrow down* the origin of a given source code artifact to a relatively small set of candidates (ideally one) when searching within the Python open-source ecosystem?

To answer this question, I designed and executed a series of experiments in which a certain number of products were sampled as subjects from the corpus. In the experiments, I used a variable number (1 to 5) of identifier(s) extracted from every subject to find the relationship between the number of identifiers used and the number of candidates returned. The results showed that, on average, using 3 identifiers can narrow down the size of the set of candidates to less or equal 3 products, with a probability of 89%.

Lastly, I evaluated the approach by trying to match Python packages that are outside the reference corpus, thus the third research question proposed is:

> **RQ 3**: Is the proposed approach useful when the subject artifacts are *outside* the reference corpus (i.e., the PyPI ecosystem)?

To answer this question, I applied the approach to determine the origin of Python packages within the Debian GNU/Linux distribution. I selected 2,174 Python packages distributed in the Debian 10 Buster as the experimental subjects, knowing that the same products have also been distributed in the PyPI ecosystem. I devised a simple method to select the origin among the candidates depending on an OSS ranking metric named SourceRank. With a fixed signature size of 3 identifiers, the results are as follows:

1. In 55% of the times, only one candidate was returned, which was the exact origin.

2. In 33% of the times, two or more candidates were matched, and the origin was one of them. For which, by ranking the candidates using SourceRank, the origin was at the top of the candidates in 68% of the times.

3. In the rest 13% of the times, the approach failed since the subjects contained identifiers that were not found in the PyPI prodcuts.

1.4 Contributions

The main contributions made by this thesis can be summarized as follows:

1. **An corpus of identifiers from the PyPI ecosystem**. For this study, I obtained the source code of 1.8 M releases from 244 K software products distributed by PyPI, then extracted 2.7 M class names and 8.5 M function names

from Python source files to build the corpus. The corpus is used for querying entities where a given identifier (or set of identifiers) is defined by looking up <*identifier, entity*> instances, in which there are three types of entities by hierarchical levels of granularity: product, release, and file. The number of instances is 26.8 M for the level of product.

2. **An empirical study of the distribution of the distinctiveness of global identifiers in the PyPI ecosystem**, which provides a theoretical basis for the proposed approach. The main result was that, in the PyPI corpus of 244 K products, there exist 11 M identifiers, and 76% of these identifiers are found in one product, and about 93% of the identifiers are found in at most 3 products.

3. **An approach to determine the potential origin of source code artifacts**. I devised an algorithm to find copies of Python source code artifacts with the corpus by taking advantage of the distinctiveness of identifiers. Given a set of identifiers extracted from an artifact as input, the approach finds a list entities from the list that are the potential copies of the artifact. Additionally, I found that the 300 most frequently appeared identifiers in the corpus are relatively not helpful as input; thus they should be regarded as stop-words and excluded from being sampled.

4. **An evaluation of the approach**. I evaluated the approach by matching external source code artifacts from a "golden" set of 2,181 Python packages distributed by Debian GNU/Linux ecosystem as the experimental subjects. I tried to find these products with the proposed approach: take 3 identifiers from each subject that are then matched against the PyPI corpus, to find product-level candidates of the subjects. The results showed that in 87% of the times, the actual origin of the subject was in the returned candidates. In addition, I sorted the candidates relying on a product popularity and quality metric named SourceRank, so that for 68% of the times, the subject's origin appeared at the top of the candidates when more than one candidates were found.

1.5 Thesis Organization

This thesis is organized as follows.

Chapter 2 contains the required background knowledge for understanding the software origin problem and shows the current research stage on the problem. I also summarized related research about the nature and usage of identifier names and the alternative yet conventional origin determination methods.

Chapter 3 describes the proposed conceptual model, including the definition of a reference corpus, the distinctiveness of identifiers, and a method that matches the identifiers to an ordered list of origin candidates.

Chapter 4 covers the implementation details of the model, including source code & SourceRank retrieval, identifier extraction, database implementation, and the matching algorithm of the method.

Chapter 5 demonstrates the empirical studies performed to answer the first two research questions, including the analysis of PyPI products, the properties of the identifiers (e.g., the distribution of distinctiveness and the frequent identifiers), and an experiment to determine the number of identifiers required to obtain a satisfied number of candidates.

Chapter 6 reports another experiment for answering the third research question, which evaluated the method by matching 2,181 Python packages from Debian to products in the PyPI corpus.

Chapter 7 provides further discussions on the results, insights drawn from the empirical studies, limitations of the method, future studies, and a thesis conclusion.

Chapter 2

Background and Related Work

This chapter includes descriptions of the background and previous studies related to this thesis, which is organized into two parts. First, I will discuss why identifiers interest researchers and how they play an essential role in solving software engineering problems. The second part comprises the research on the provenance problem in software development processes and other common techniques that can also work as alternatives to the proposed method of origin determination and issues that can further be addressed with the origin information.

2.1 Source Code Identifiers

Identifiers are the lexicons that declare certain entities in source code. Identifiers account for one-third of the tokens and over 70% of the characters in source code [17]. There has been a vast research interest in the patterns and semantics of identifier names, which somehow explains why identifiers are commonly distinct and can be used to retrieve software artifacts. Here I outline the works that inspired the development of my approach.

There is a consensus among researchers that identifiers names play an important role on code comprehensibility, maintainability, and overall quality [12][40][17][39][41] [19][10][8][70]. As an interesting fact, it was common in the 90's that programmers in legacy systems naming identifiers arbitrarily after their girlfriends, favorite sportsmen, etc. [71]. There have been many studies on lexical patterns, quality metrics, and models of identifiers. For example, Deissenboeck and Pizka [17] proposed rules for creating consistent and concise identifier names, which involved maintaining an

identifier dictionary alongside the software development. Lawrie et al. [41] found that both words and abbreviations could lead to good comprehension, but excessively long identifiers overload short-term memory. Binkley et al. [8] claimed that the style of identifiers, such as abbreviation and `camelCase`, has a tremendous impact on program understanding, quality, and cost. Hofmeister et al. [26] found shorter identifier names take longer to comprehend; using words as identifier names helps to improve software quality and save costs. Overall well-constructed names can improve comprehension activities by 19%. By conjecture, identifier names in modern software development tend to be distinct since they are required to precisely describe code entities with diversified functionalities, plus various styles to use.

It is noticeable that there has been a lot research efforts to develop techniques for suggesting good names or detecting bad names based on semantics in code entities [29][5][46][47][58][62]. Particularly, Lacomis et al. [38] proposed a method that predicted identifiers from compiled binaries via reverse engineering.

Identifiers are used to solve various kinds of problems in software engineering, such as detecting bugs [65][67][66] and vulnerabilities [25], as well as completing partial code [60]. To the point of this thesis, the trend of higher quality identifiers implies that identifiers can potentially be used in solving searching-related problems. The intuition is that when identifier names explicitly convey their semantics and entity functionalities, they tend to become unique. Nguyen et al. [58] found that 62.9% of the method names are unique among a selected set of quality Java projects. Holmes et al. [72][31] proposed a tool named Baker, which integrated information about a library used in source code — including its API documentation, code reviews, issues, related Q&As, etc. The tool used identifiers as the medium and adopted a similar approach as this thesis: reducing the number of candidate matches by sequentially querying the extracted identifiers.

2.2 Origin of software artifact

The problem of origin determination has also been researched as finding the *provenance* of software artifacts. Generally, the term provenance denotes the lineage of an artifact, including its origin, history, ownership, movement, etc. In the past two decades, provenance has been used in the software development process (SDP) context. By definition from the World Wide Web Consortium [77], "provenance is information about entities, activities, and people involved in producing a piece of data or

thing, which can be used to form assessments about its quality, reliability or trustworthiness." In a narrowed scope, regarding a *software development artifact* (such as a feature, component, chunk of code, test suite, etc.), developers, managers, QA team members, and other stakeholders often wish to figure out how and why it came to be where it is [22]. The provenance of development artifacts can be used to comply with security standards, licensing, and other software requirements [16].

In the remaining section, to provide an approximate overview of the topic, I will first introduce some studies in the specific domain of provenance in SDP, including generic data models and techniques used to capture and store provenance data of development artifacts. Then, I will show studies of conventional yet popular techniques (such as code clone detection) that can also be used to determine the origin of software artifacts at some point but are often too expensive to perform at a large scale. Some origin-related problems will also be discussed.

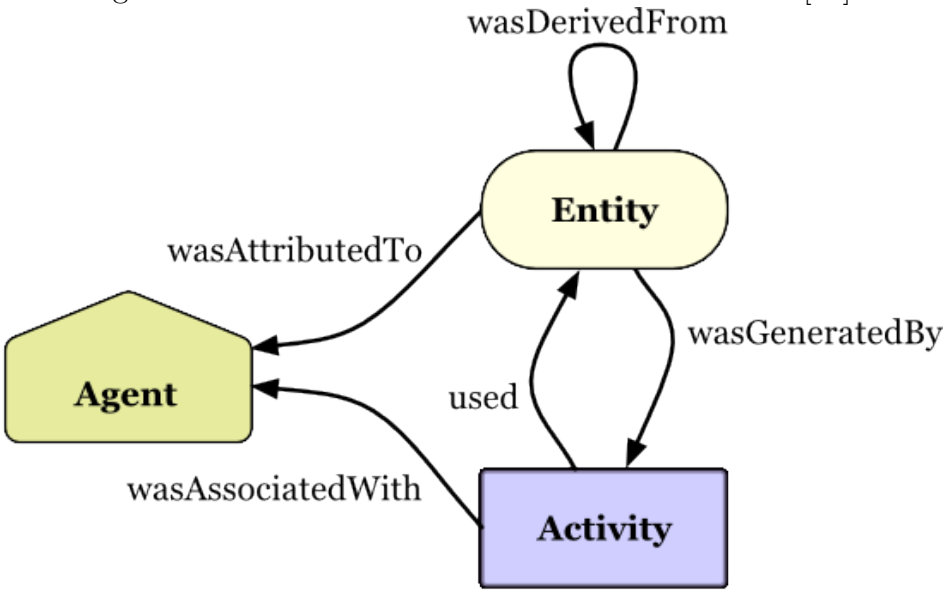
2.2.1 Provenance in software development process

Dalpra [13] did a systematic review in the area of provenance in the context of SDP. She pointed out the characteristics of the existing studies, as the area is still immature, there was no research before 2005, the goals are very different, there is no consensus about the most suitable model, and empirical studies are scarce.

Researchers focus on quite specific aspects in this area. The first aspect that has received much attention is the *data models* used to capture and store provenance data of generic objects. Sun et al. [73] pointed out that provenance data is an immutable directed graph incrementally captured at run-time, so it differs from traditional metadata. Two mainstream models are used to capture and store provenance data: the Open Provenance Model [55], and PROV [56]. PROV is a family of specifications to express, model, access, exchange, serialize, and reason over provenance records [52]. It expresses provenance data through the description of entities (physical, conceptual, digital, etc.), web documents, activities (actions that create or use entities), agents (people or objects that are responsible for some entity or activity), and the relations between them. Figure 2.1 depicts the basic structure of the PROV model. PROV model has been specialized to specific standards and domains, such as Prov-dm[7], D-PROV[53], ProvOne[11], PROV-Process[14], and Versioned-PROV[63]. The PROV family keeps expanding and is widely adopted by emerging provenance frameworks. For example, Bose et al. [9] propped Blinker, a framework enabled by blockchain

technologies and the standard of PROV-SWProcess.

Figure 2.1: The basic structure of the PROV model [76]



Another essential aspect is typically focusing on *provenance of development artifacts*, which is also the topic of this thesis. Xu and Sengupta [79] did the earliest research and proposed an approach named Fully Traceable System that binds a general artifact to its traceability and evolution information in configuration management. Dang et al. [15] proposed Ariadne, an IDE-based provenance management tool for tracking the provenance of source code and generating provenance reports via observing potentially risky events such as pasting, to help the management of intellectual properties. Davies et al. [16] proposed Software Bertillonage: the term is a metaphor for fast narrowing down the enormous search scope to a tractable set of likely suspects, then applying more expensive techniques to identify the final origin. They implemented an anchored class signature technique to find occurrences of a source code entity based on computing the similarities between classes, similar to this thesis. However, the method proposed by this thesis uses less information from the entity (only a few random identifiers) to comparably achieve the same goal. Godfrey [22] described the problem of extracting and reasoning about the provenance of software development artifacts and other provenance-related problems such as clone/IP violation detection, recommender system, bug report duplication, etc. Rousseau et al. [69] exploit the possibility of tracking software provenance at the largest scale ever by analyzing the replication factor of development artifacts (e.g., SLOC, files, and

commits) within the Software Heritage archive.

2.2.2 Alternative methods and related problems

The problem of determining the origin of source code artifacts has conventionally been addressed by various other approaches, such as code search and clone detection. These areas and my proposed work have quite strong relations with each other because they often comprise shared purposes and techniques.

Code search is generally the practice of querying for code examples with free-form input, either in natural language or code, within a code base. Early approaches [44][51][49][50] usually regard source code as textual documents and adopt information retrieval models to find code snippets that match a given query relying on the textual similarities, but lack semantic mappings between code and queries [24]. There are recent studies that have leveraged the semantics in source code. For example, Kim et al. [36] proposed FaCoY, a code-to-code search engine that identifies tokens that are alternatives of which in a given input code fragment and matches to snippets in Q&A posts that have similar descriptions. Also, Gu et al. [24] applied deep learning to measure the semantic similarities between natural language or code queries and code snippets.

Code clone detection is the practice of determining if code has been intentionally copied from somewhere else. Clone detection can be widely applied in many scenarios, such as plagiarism detection, malware analysis, software vulnerability assessment, and program comprehension [30]. Like code search, most existing approaches focus on textual, structural, and syntactic clones. In contrast, a limited number of approaches focus on semantical clones, even though the field has been active for a long time [36]. Accordingly, these common approaches were designed as token-based [33][43][80], Abstract-syntax-tree-based [6][32], and dependency-graph-based [37][45]. Typically, some researchers also proposed approaches relatively simple token-based approaches that are similar to the proposed identifier method. For example, Di Penta et al. [18] used filenames and class names to identify software licenses, and Ossher [59] analyzed file-level clones by applying several trivial methods: exact file matching, filename matching, identifier name fingerprint matching, and directory matching. More recent studies also adopted machine learning techniques to detect clones [78], including cross-language clones [61].

Moreover, other popular research fields can take advantage of the techniques in

origin determination, code search, and clone detection. For example, *code recommendation systems* [27][28][23][64] can prompt coding with reusable code from sources such as open source repositories or Q&A sites, and they are often enabled by IDEs. Likewise, *program repair* [42][20][34][48] is about fixing buggy code by replacing it with similar code in some cases from public sources, and *API recommendation systems* [72][31][57] can analyze code snippets and find relevant API documentations and coding examples.

Chapter 3

Conceptual Model

This chapter introduces the conceptual model of the proposed approach for determining the origin of a given software artifact. First, it describes the structure of a reference corpus used to match against, followed by the definition of the key property of identifiers, namely *distinctiveness*, which measures how unique an identifier is in a corpus. Then, it illustrates the algorithm for finding the origin of a subject entity within the corpus and how to select the correct origin when multiple candidates are all likely to be the origin.

3.1 Reference Corpus

First, one needs to specify in which reference scope (i.e., an OSS ecosystem) they wish to find the *origin* of a *subject*. Thus, our matching algorithm requires a *corpus* that is built from downloading and processing (i.e., extracting *identifiers* from) the entire body of source code of the software artifacts in the ecosystem. The corpus contains the mapping from identifiers to source code *entities* where the identifiers are defined. The corpus is also ideally built for an ecosystem that is as comprehensive as possible, such as a version control repository (e.g., GitHub or GitLab) or a source package management platform (e.g., the Python Package Index (PyPI), the Comprehensive R Archive Network (CRAN), Maven Central for Java packages, NPM for JavaScript packages, etc.).

From the paragraph above, the nomenclatures are defined as:

- **Entity**: The body of a software artifact that includes source code. Entities in a software body exist in a hierarchical (i.e., recursive) structure, such that higher-

level entities are divisible into lower-level entities. In this thesis, for example, a top-level entity is a software *product* that includes all the existing versions of source packages it has released over time, while a bottom-level entity is a single source file that is contained in a specific release. With this property, we can define the *level of granularity* of the matching (i.e., how *fine* the origin's location is reported), which will be discussed with more details in the rest of this section. An entity is defined by its *name* (e.g., name of product, version number of release, path of source file) and *type* (i.e., level of the entity).

- **Identifier**: A lexical token created in the development process to declare an artificial component in source code such as a class, a function, a parameter, a variable, etc. In this thesis, an identifier is defined by two attributes: its *name*, and the *type* of the component it declares. Since the algorithm, to be efficient, relies on minimum information of the source code, I only focus on identifiers of global *functions* and global *classes* (and their methods, also regarded as functions)[1]. In the rest of the thesis, I will use the terms *class* and *function* to refer global class and function (including class method) names.

- **Corpus**: A reference corpus is where the matching algorithm finds the copy of an entity representing the body of a large OSS ecosystem. A corpus has three parts of data: 1) the entities, 2) the identifiers, and 3) the mappings between 1) and 2). In this thesis, the corpus is built from the Python source code of all the 244,084 products (as of August 2020) distributed in the PyPI ecosystem.

- **Subject**: The entity outside the corpus whose origin is being investigated. In this thesis, a subject is either a file or a set of files.

- **Origin**: The entity inside the corpus which is an identical or almost identical copy of the subject, asserted by the algorithm, from where the subject appears to have originated.

As mentioned, entities are classified by the hierarchical structures of software artifacts. For a corpus defined by the algorithm, the following three types of entities are considered:

[1]By the term *global*, I consider functions and classes that are defined at the top level of modules, so that they can be imported by other modules, or potentially copy-pasted, to external modules

- **Product**: A software entity that can be distributed to users and deployed as a whole. Products can be software projects, dependencies, applications, development frameworks, etc. In this thesis, products are code of packages distributed in the PyPI ecosystem. A product is defined by its *name*, which is unique in a corpus.

- **Releases**: Releases are versions of code that a product has distributed over time. A release is defined by its *product* and *version number* (e.g., `foo-1.0.0`).

- **File**: A single source file that are contained by a release, defined by the *release* and *file path* (e.g., `foo-1.0.0/some/path/bar.py`).

Formal notations are used to illustrate the composition of the corpus. Let $Defs(e)$ denote the **set** of identifiers (without any duplication, same as below) defined in entity e (an entity of a specific type E) that is in a corpus, and there are:

- $Defs(f)$ denotes the set of identifiers defined in file f.

- $Defs(r)$ denotes the set of identifiers defined in release R, which is the union of $Defs(f)$ for all the files in r:

$$Defs(r) = \cup_{F_i \in F} Defs(F_i)$$

  where F is the set of files in r.

- Similarly, $Defs(p)$ denotes the set of identifiers defined in product p, which is the union of $Defs(r)$ for all the releases in p:

$$Defs(p) = \cup_{R_i \in R} Defs(R_i)$$

  where R is the set of releases in p.

Note that, the algorithm does not rely on the *frequency* that an identifier appears in the same entity. Instead, the algorithm only determines whether the identifier is present or absent. Therefore, the definitions above use **sets** to describe the containment of identifiers.

Moreover, with the information above, entities in a corpus are divided into 4 *collections* so that the origin can be determined at respective levels of granularity. C_E denotes a collection of entities for a given type E that is one of **P**roduct, **R**elease,

or **F**ile. I also define a special collection, the **L**atest **R**elease of each product, denoted as C_{LR}, which is a subset of C_R. Figure 3.1 shows an example of building the four collections from a corpus C with two products P_0 and P_1, where $P_0 - R_1$ is the latest release of P_0, and $[a, b, c, d]$ are identifiers.

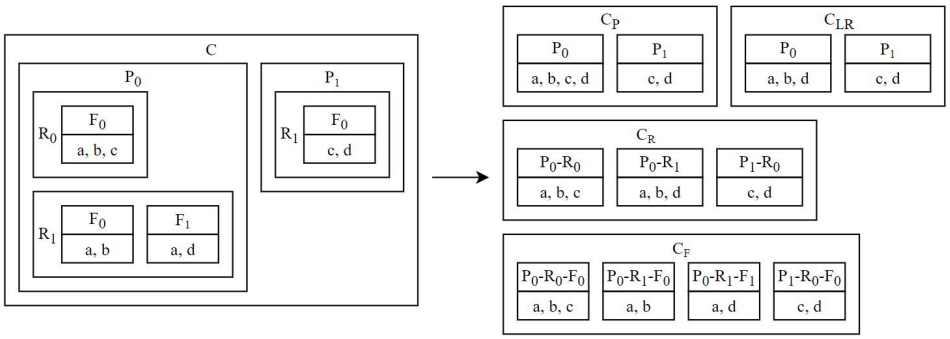


Figure 3.1: An example of building collections in a corpus

3.2 Distinctiveness of Identifiers

Recall that our approach is based on the conjecture that the overall identifiers are highly unique in a reference corpus even built from a large body of OSS ecosystem. Therefore we can find the origin (or a minimal number of candidates) with very few identifiers extracted from a subject. To verify this conjecture as well as estimate the effectiveness & efficiency of the algorithm, we need to quantify and analyze *how unique* the identifiers in a corpus are.

Hence I defined *distinctiveness*, which is regarded as the key property of an identifier. Distinctiveness measures *how many* entities in a collection that define a given identifier. Formally, the distinctiveness for identifier id respects to a collection C_E is denoted as $dist(id, C_E)$, which is formally defined as:

$$dist(id, C_E) = |\{e | e \in C_E \wedge id \in Defs(e)\}|$$

There are two special cases of distinctiveness: the identifier id is not defined by any entity in C_E iff. $dist(id, C_E) = 0$; and id is *unique* in C_E iff. $dist(id, C_E) = 1$, which means that in C_E, there is only one e that defined id. The lower distinctiveness value implies that the identifier is more distinct so that the identifier produces higher selectivity in matching results (i.e., returns fewer candidates). The highest possible

distinctiveness value is the total number of entities in C_E, in which case id appears in every entity.

Figure 3.2 shows an example of the distinctiveness of four identifiers for two collections.

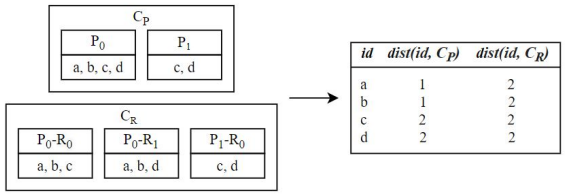


Figure 3.2: Example of distinctiveness respect to two collections

The next chapter describes how this model was implemented using PyPI as a corpus. In chapter 6 I describe an empirical evaluation of the effectiveness of this implementation.

3.3 Matching Identifiers

For a given subject, the algorithm finds its origin within the corpus in the following way: a small set of identifiers are extracted from the *subject* entity, forming a *fingerprint* of the subject. At a specified level of granularity, the algorithm returns a relatively small set of entities in the corpus that have defined **all** of the identifiers in the fingerprint. These entities are regarded as the *candidates* of the origin.

Hence, the following function is an abstraction of the matching procedure:

$$matching(s, C_E) = CS_E$$

where:

1. Parameter s is the subject entity. In this thesis, a subject is a file or a set of files.

2. Parameter C_E is the collection of entities of type E in the corpus, where the origin will be searched.

3. Return value CS_E is the **set** of candidates. Each candidate is an entity of type E, which contains all the identifiers in the fingerprint generated from s.

It is worth noting that there is an intermediate procedure in *matching*, which is creating the fingerprint from the subject with a sampling function:

$$sampling(s, conf) = fingerprint$$

where $conf$ is the configuration of the sampling, possibly containing:

1. The number of identifiers in the fingerprint.
2. A list of frequent identifiers in the collection that are avoided from being sampled.
3. (If the subject is a set of files) whether to sample the identifiers from one file or to sample each identifier from a disjoint file.

Details of the sampling algorithms will be described in Section 4.2.

3.4 Ordering the set of Candidates

Although the algorithm is expected to return a small number of candidates, we need a method to select one from these which is to be reported as the best potential origin of the subject. Instead of applying exhaustive approaches evenly for every candidate, it might be helpful to order these candidates so that candidate with a higher possibility of being the origin will be investigated sooner. The sorting is most practically applied at the coarsest level of granularity: product.

The initial idea was to create a product evaluation index based on the products' popularity and quality measurements in the PyPI ecosystem corpus. However, it was hard to complete since PyPI does not provide the necessary information, such as the download count of its released packages. However, I found SourceRank to be ideal for these requirements, a metric that Libraries.io[2], uses for sorting its search results. SourceRank measures products' properties of 5 categories: code, community, distribution, documentation, and usage. The overall score is computed by summing up the subscores from each category, meanwhile, each subscore is either a binary value or a number (possibly subjected to logarithmic or scaling). Figure 3.3 shows the subscores of the product with the highest SourceRank in PyPI, `requests`[3]. For

[2]A compositive OSS index site covering products distributed by 32 known repositories, including PyPI. `https://libraries.io/`, last accessed on 2022-10-13

[3]`https://libraries.io/pypi/requests/sourcerank`, last access on 2022-10-13

example, subscore *Dependent Packages* accounts for how many other products are dependent on this product, and the value is computed by taking the logarithm of the original number and then multiplying by 2.

Figure 3.3: The SourceRank subscores of product `requests`

Basic info present?	1
Source repository present?	1
Readme present?	1
License present?	1
Has multiple versions?	1
Follows SemVer?	1
Recent release?	1
Not brand new?	1
1.0.0 or greater?	1
Dependent Packages	8
Dependent Repositories	5
Stars	5
Contributors	2
Libraries.io subscribers	2
Total	31

In this way, a refined matching function is defined, which returns an ordered **list** (instead of set) of candidates, as follows:

$$matching(s, C_P, SourceRank) = CL_P$$

where C_P is the collection of products, and CL_P is the list of candidate products that have been subjected to ordering with SourceRank score for these products.

3.5 Summary

This chapter has described the conceptual model for determining the origin of a software artifact. It includes 1) what is and how to build a reference corpus; 2) the definition of the distinctiveness of an identifier with respect to a collection of entities in a corpus for measuring how frequently the identifier is defined; 3) matching the subject to a set of candidates of the origin from a collection of entities in the corpus, and 4) sorting the candidates based on SourceRank so that the most likely origin can be reported.

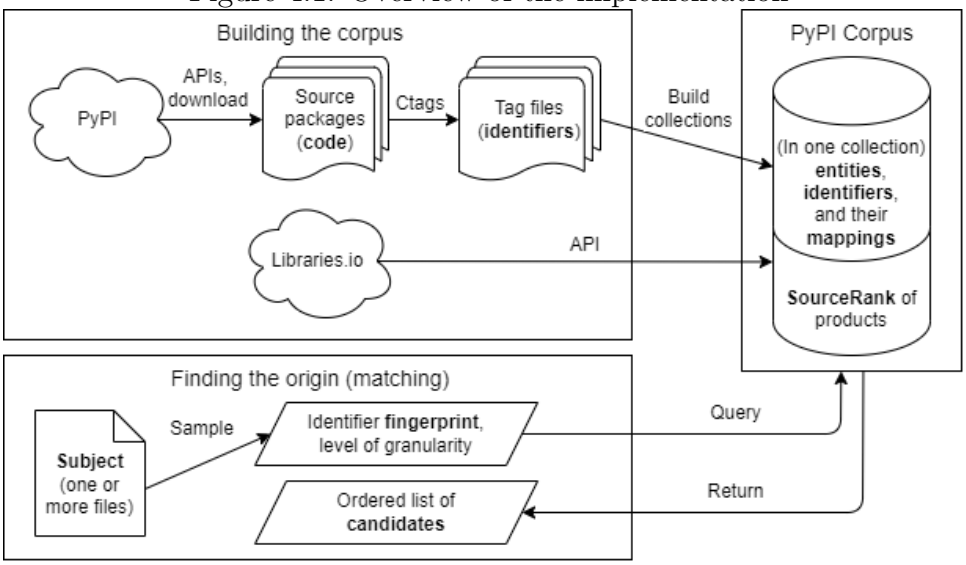
Chapter 4

Implementation

This chapter explains how the conceptual model was implemented, mainly about the process of building a corpus from a massive number of software packages distributed in the PyPI ecosystem. The primary and heaviest works were retrieving source code from PyPI repositories and extracting identifiers from the code entities. Then, the identifiers, the entities, and their mappings were stored in a database separated by the entity types, serving as entity collections where the algorithm finds the origins. Also, the SourceRank for each product is retrieved using the API provided by Libraries.io. The rest of this chapter describes the procedure of matching a subject entity against the corpus, including creating an identifier fingerprint (i.e., a small set of identifiers) from a subject and querying it in a collection of entities.

Figure 4.1 shows an overall picture of the methodologies introduced in this chapter. The top-left block shows the procedures of retrieving and processing the data to build the corpus, the block on the right-hand side shows the contents of the corpus, and the block on the bottom shows how to use the corpus to find the origin of a given subject entity.

Figure 4.1: Overview of the implementation



4.1 Building the Reference Corpus for PyPI

I started building the PyPI corpus on August 20, 2020. At that time, PyPI claimed to have 257,466 projects (i.e., products), 2,027,249 releases, 3,171,105 files (downloadable packages), and 447,062 users[1]. I took advantage of the cloud platform Compute Canada's Cedar cluster[2] for downloading, storing, and processing the source code. I used the tool Slurm[3] to split the workload and execute as many as 100 jobs in parallel, each by one CPU core. The entire execution of building the corpus took about 5 days.

4.1.1 Retrieving the source code

I retrieved the source code from PyPI using the following method:

The first step was to list all the products with PyPI's official Simple Repository API (PyPI-simple)[4], a package index that is intended for automated usages. PyPI-simple provides a web API[5] that lists all the products and their package files. I crawled the HTML document returned by the API to get the names of the 257,466 products.

[1]A snapshot of PyPI's homepage on August 20, 2020 can be found at Wayback Machine: `https://web.archive.org/web/20200820134656/https://pypi.org/`, last accessed 2022-10-26
[2]`https://docs.computecanada.ca/wiki/Cedar`, last accessed 2022-10-26
[3]`https://docs.computecanada.ca/wiki/Running_jobs`, last accessed 2022-10-26
[4]Defined in PEP 503: `https://peps.python.org/pep-0503/`, last accessed 2022-10-26
[5]`https://pypi.org/simple/`, last accessed 2022-10-26

The second step was obtaining releases from each product and links to download the packages. For this, I used another official tool of PyPI, the PyPI-JSON API[6] that is used to query the metadata of a product by its name. As the example shown below, the API returns the data in a JSON string[7], containing all the information that can be found on the corresponding product webpage[8]: the description, the author, the external repositories, the release history, the package files, etc.

[6] `https://warehouse.pypa.io/api-reference/json.html`, last accessed 2022-10-26
[7] `https://pypi.org/pypi/sampleproject/json`, last accessed 2022-10-26
[8] `https://pypi.org/project/sampleproject/`, last accessed 2022-10-26

```
{
    "info": {
        "author": "The Python Packaging Authority",
        "author_email": "pypa-dev@googlegroups.com",
        ...
        "home_page": "https://github.com/pypa/sampleproject",
        "keywords": "sample setuptools development",
        ...
        "name": "sampleproject",
        ...
        "project_url": "https://pypi.org/project/sampleproject/",
        "project_urls": {
            "Download": "UNKNOWN",
            "Homepage": "https://github.com/pypa/sampleproject"
        },
        ...
        "summary": "A sample Python project",
        ...
    },
    "last_serial": 1591652,
    "releases": {
        "1.0": [],
        "1.2.0": [
            {
                ...
                "filename": "sampleproject-1.2.0-py2.py3-none-any.whl",
                "has_sig": false,
                "md5_digest": "bab8eb22e6710eddae3c6c7ac3453bd9",
                "packagetype": "bdist_wheel",
                "python_version": "2.7",
                "size": 3795,
                "upload_time_iso_8601": "2015-06-14T14:38:05.093750Z",
                "url": "https://files.pythonhosted.org/packages/30/52/547eb3719d0e872bdd6fe3ab60ce
                f92596f95262e925e1943f68f840df88/sampleproject-1.2.0-py2.py3-none-any.whl",
                ...
            },
            {
                ...
                "filename": "sampleproject-1.2.0.tar.gz",
                ...
                "packagetype": "sdist",
                ...
            }
        ]
    },
    ...
}
```


However, I found PyPI-JSON did not return any information for 13,382 products (it returned 404 errors instead). By manual inspections, these products seemed to no longer exist on the PyPI website, and neither were downloadable packages returned by PyPI-simple. Therefore, the number of products the corpus would include was 244,084 (257,466 - 13,382).

The third step was to download the releases of each product via package file links provided by the JSON data. Most products had few releases, as the median number of releases per product was 7.5, with a standard deviation of 13. Meanwhile, I noticed that a few products had excessive numbers of releases. For example, a cryptocurrency library named `CCXT`[9] had released 7,400 times during its age of 3 years; however,

[9] `https://pypi.org/project/ccxt/#history`, last accessed 2022-10-26

the updates mostly happened in configurations of currency exchanges rather than in source code. It was considered non-necessary to download all of these releases since they would cost too much time to execute and too much space to store; instead, I decided to retrieve at most 100 recent releases of every product. This way, the number of impacted products (with more than 100 releases) was only 1,451, accounting for 0.6% of all the products.

Moreover, a release often has more than one package file. In PyPI's packaging system, there exists *source distributions* (`packagetype = sdist` in metadata) and *binary distributions* (`wheel` and `egg` formats, `packagetype = "bdist_wheel"` or `"bdist_egg"`). Ideally, I downloaded source packages since they contain the complete source code. A source package is usually offered in multiple archive file formats (e.g., `.zip`, `.tgz`, `.bz2`, etc.) with the same data, therefore, I only needed to download one of the files, and I chose the first one listed. However, I downloaded the first binary package when a release did not provide a source package. Because Python is an interpreted (rather than compiled) language, binary packages still contain source files necessary to run the software, while possibly the test or build files are not included. The following procedure restates how I selected the package files.

```
if the release has any source distribution then
    download the first source distribution in JSON
else
    if the release has any binary distribution then
        download the first binary distribution in JSON
    end if
end if
```

At the end of this phase, I downloaded 1,831,172 package files, one for each release. These packages (mostly compressed) took 1.6 TBytes of disk space. Table 4.1 documents the counting of each downloaded packages type. It shows that 90% of the packages were source distributions. 132 binary distributions (sum of last five rows) were neither wheel nor egg packages; instead, they were installers for different operating systems (e.g., `.exe` for Microsoft Windows) and did not come with any source code. I kept these 132 releases in the corpus while marking them empty.

Table 4.1: Statistics of package files by distribution types

File extension	Package type	Count
.gz	sdist	1,575,511
.whl	bdist_wheel	180,377
.zip	sdist	64,017
.egg	bdist_egg	7,599
.bz2	sdist	3,341
.tgz	sdist	195
.exe	bdist_wininst	73
.rpm	bdist_rpm	51
.msi	bdist_msi	4
.deb	bdist_dumb	3
.dmg	bdist_dmg	1
Total		1,831,172

4.1.2 Extracting identifiers from source code

I decompressed each package to access the source (`.py`) files using `UnZip` or `tar` tool, depending on the package file format. I tested multiple methods for extracting the identifiers from Python source code, such as regular expression and Abstract Syntax Tree (AST) parsing. Finally, I decided to leverage the tool Universal **Ctags**[10] because I found it reliable, efficient, and easy to use.

Ctags can be run as a bash tool and generates indexes (called tag files) for code components (e.g., classes, methods, functions, parameters, variables, etc.) defined in source files written in various programming languages. These indexes enable IDEs and other tools to locate the indexed identifiers in the code. One can also execute Ctags at the top level of a package directory (with flag `-R` that stands for recursively). The locations of identifiers in the whole package will be output into one tag file. Hence, I could conveniently use Ctags to list file entities in a release and extract identifiers from each file.

Figure 4.2 shows a sample tag file created for a downloaded release of product `NumPy`. Each tag file has five columns, from left to right: identifier name, identifier

[10] `https://ctags.io/`, last accessed 2022-10-26

type, the line number where the identifier is defined, the path of the source file, and the actual code line. Note that identifier type `member` refers to class method, which is regarded as a function in this thesis. Also, I only recorded classes and functions that came from Python source files[11]. However, if a Python source file had no classes or functions, I would not store the file entity into the corpus.

Figure 4.2: A sample tag file (fragments) output by Ctags (with flags -R and -x) for a release of product NumPy

```
reshape  function  200   numpy/core/fromnumeric.py         def reshape(a, newshape, order='C'):
reshape  function  7143  numpy/ma/core.py                  def reshape(a, new_shape, order='C'):
reshape  member    4593  numpy/ma/core.py                  def reshape(self, *s, **kwargs):
...
sctypes  variable  225   numpy/core/_type_aliases.py       sctypes = {'int': [],
sdist    class     9     numpy/distutils/command/sdist.py  class sdist(old_sdist):
```

I noticed that 13,739 (5.6%) products did not include any file entities (i.e., Python files with at least one class or function)[12] therefore, they were not associated with any identifiers in the corpus. I identified that these products were either empty (at least for software purposes) or written in languages other than Python. Table 4.2 documents numbers of entities and identifiers in the corpus.

Table 4.2: Descriptive Statistics of the entities and identifiers in the PyPI corpus

Item	Count
Products	244,084
Releases	1,831,172
Files	45,239,359
Class names	2,665,927
Function names	8,598,979

4.1.3 Retrieving the SourceRank of products

Like PyPI-JSON, Libraries.io provides its official API[13] for querying the SourceRank of a given product in a given ecosystem so that it was straightforward to retrieve the

[11] since a package might contain source files in other languages that Ctags detect, I ignored entries that were not from .py files

[12] including products that did not offer any source, wheel, or egg packages, as discussed before.

[13] `https://libraries.io/api`, last accessed 2022-10-26

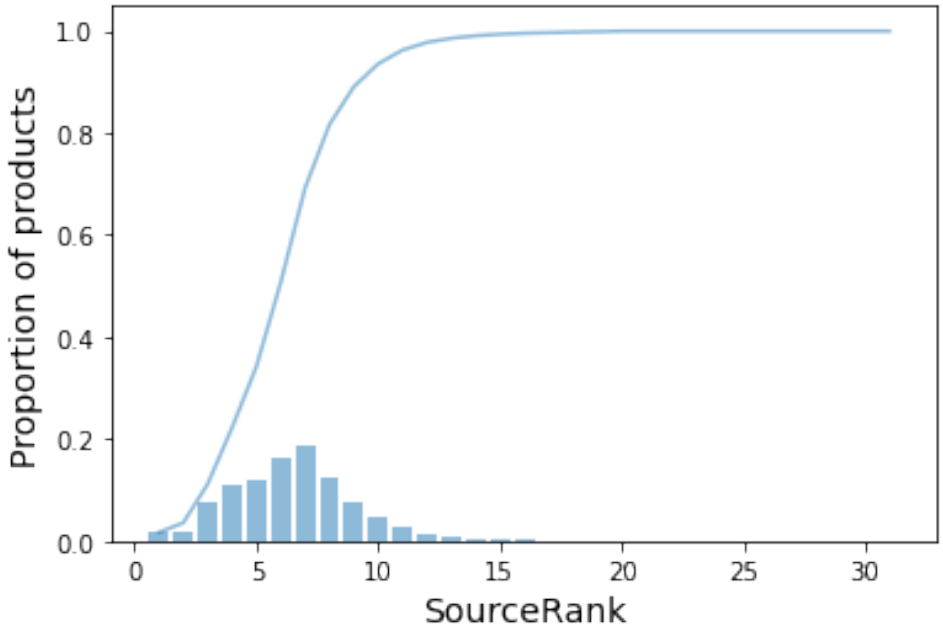
SourceRank of each product that I had downloaded.

Nevertheless, there were 479 products missed by Libraries.io, so I set their SourceRank to 0. Finally, table 4.3 documents 10 products in the corpus with the highest SourceRank, and Figure 4.3 shows the distribution of SourceRank for the entire corpus (the mean SourceRank is 6.5, and the median is 6).

Table 4.3: Top 10 SourceRank in the PyPI corpus

Product	SourceRank
requests	31
Django	30
numpy	29
Sphinx	29
Flask	28
Jinja2	28
matplotlib	28
pandas	28
pytest	28
click	27

Figure 4.3: The distribution of SourceRank in the PyPI corpus (the curve indicates the cumulative distribution)



4.1.4 Storing in a database

In the previous steps, I extracted entities, identifiers, and SourceRank covered by the corpus. I implemented the data storage using an SQLite database that could efficiently serve a corpus with such a volume, and the detailed table structures are as follows.

First, I defined three tables for the three types of entities: product, release, and file. I maintained inclusion relations between entities, as shown below:

Product

Attribute	Data type	Description
pid	Integer	Product ID, primary key
pname	String	Product name
rank	Integer	SourceRank

Release

Attribute	Data type	Description
rid	Integer	Release ID, primary key
pid	Integer	Product ID, foreign key
ver	String	Version number for the product
is_latest	Boolean	Flag for the latest release
pkg_path	String	Path of the downloaded package (optional)

Note that I identified latest releases by comparing the version numbers with Python module `packaging.version`[14].

File

Attribute	Data type	Description
fid	Integer	File ID, primary key
rid	Integer	Release ID, foreign key
fname	String	File name (path in package)

I stored the identifiers and their relations to entities in tables three tables for three collections of entities, as below:

CollectionProduct

Attribute	Data type	Description
iname	String	Identifier name
type	Boolean	Identifier type represented in binary
pid	Integer	Product ID, foreign key

CollectionRelease

Attribute	Data type	Description
iname	String	Identifier name
type	Boolean	Identifier type represented in binary
rid	Integer	Release ID, foreign key

[14] `https://pypi.org/project/packaging/`, last accessed 2022-10-26

I did not create a dependent table for the latest release collection.

CollectionFile

Attribute	Data type	Description
iname	String	Identifier name
type	Boolean	Identifier type represented in binary
fid	Integer	File ID, foreign key

This way, I stored every extracted identifier in three separate tables to create relations between the identifier and entities at all three levels where the identifier is defined. This practice added data redundancy in a relational database because (for normalizing the database) I could only store the relations between identifiers and **file** entities, then use entity relations (i.e., file to release and release to product) to enable querying at higher levels of granularity. However, the current data structure makes querying identifiers in releases and products faster by taking advantage of these "intermediate" tables and avoiding joining the entity tables.

4.2 Finding Matches of a Subject

I implemented the matching algorithm defined in Section 3.3 relying on the presented database. First, we need a method to sample a set of identifiers as the fingerprint from the subject. The size of the set (denoted by number N) plays an important role because only a small set is required (the experiment results in Section 5.3 will show that the ideal fingerprint size is 3).

For a given N, when the subject is a file, we can just randomly sample N different identifiers. When the subject is **a set of files**, we can follow either of the two sampling methods: *single-file* or *disjoint-file*. Here is the procedure to create the fingerprint using the single-file method:

1. Sample a file that has at least N different identifiers.
2. Sample N different identifiers from the selected file.
3. Return the set of identifiers.

And here is the procedure to create the fingerprint using the disjoint-file method:

1. Initiate an empty set of sampled identifiers.
2. Sample a file with replacement that has at least one identifier that are not in the set.
3. Sample an identifier that is not in the set from the selected file.
4. Repeat from step 2 until the set has N identifiers.
5. Return the set of identifiers.

Next, we can perform the matching using the fingerprint to a collection of entities:

1. Initiate an empty set of candidates, CS.
2. For each identifier in the fingerprint:
   1) Query entities in the collection that has defined the identifier.
   2) If the identifier is the first one in the fingerprint, let CS be the set of returned entities.
   3) Otherwise, let CS be the **intersection** between the current CS and the returned entities.
3. Return CS.

Lastly, suppose the origin is determined at the level of product. In that case, we can use SourceRank of the products to create a sorted list of candidates, then choose the candidate with the highest SourceRank as the finally determined origin, as follows:

1. Convert CS_P (a set of **product** candidates) to a list of candidates, LS_P.
2. Look up the SourceRank for every product in LS_P.
3. Sort LS_P by descending order of the SourceRank.
4. Return LS_P and the first product in LS_P that is hopefully the correct origin.

4.3 Summary

This chapter has described the methodologies used in the implementation. To build a reference corpus for PyPI, first, I retrieved the source code of the latest 100 releases from every 244,084 products distributed by PyPI, then extracted identifiers from the source code with Ctags; second, I gathered SourceRank of these products. I also explained the database implementation details that enable efficient matching at various levels of granularity. Lastly, I demonstrated procedures in the algorithm for origin determination, from sampling identifiers from the subject entity to matching

the identifier fingerprint to a sorted list of candidates.

Chapter 5

Empirical Analysis

This chapter illustrates the empirical analysis that I have performed, mainly to find evidence that the proposed method is both lightweight and helpful in finding origin of subject entities.

The first section describes an analysis of the PyPI corpus, which concluded that a large portion of the products in the PyPI ecosystem is relatively trivial (e.g., have few releases or are non-active). The second section describes the distribution of distinctiveness for identifiers in the corpus, showing that most identifiers are unique at the product level. The third section describes a series of experiments to determine how many identifiers are necessary to make the method helpful.

5.1 Trivial Products in the PyPI Corpus

PyPI is commonly regarded as the universal repository of Python products since it is the default source of Python's official package manager PIP. Being an open platform, PyPI does not have a strict project review process, so developers are free to publish their projects onto PyPI. This situation leads to the fact that PyPI has been used arbitrarily by developers who published empty or toy/scratch projects that have no value to the open-source community[1], and it is the primary cause of having many (5.6%) empty products (without any identifiers in any Python files) in our PyPI corpus. These products in the corpus are considered *trivial* for this research, which means they are not expected to be origins of commonly seen subject entities elsewhere.

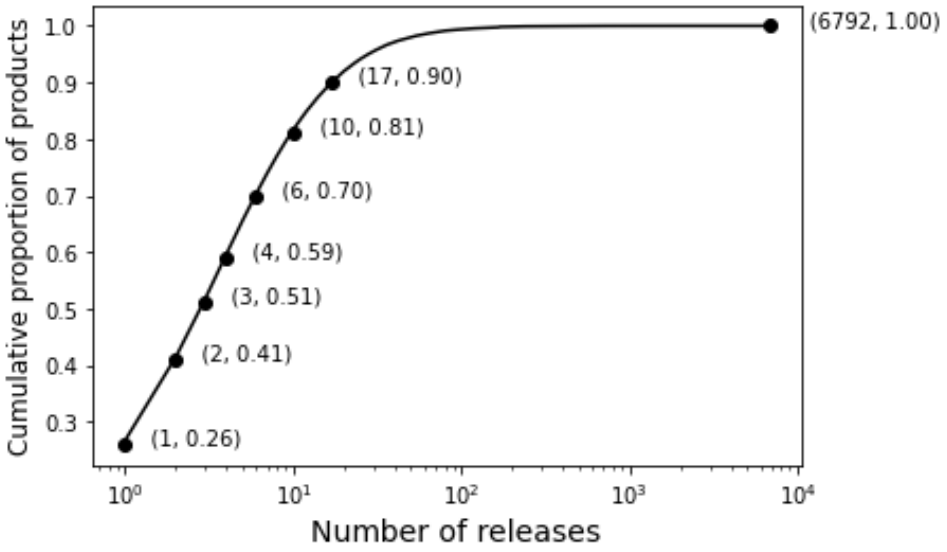
[1]This is an example of an empty project: `https://pypi.org/project/maddy5/`, last accessed 2022-11-01; many other examples can be found with "DEVELOPMENT STATUS :: 1 - PLANNING" search tag

I briefly use four product properties to estimate if it is trivial, defined in each subsection below: the number of releases, the number of identifiers, the *activeness* and *stillness*. Note that this part of the study does not intend to create a compound metric like SourceRank and determine how many products are trivial. However, the analysis heuristic might help developers and PyPI maintainers to properly use and manage this large open-source repository[2].

5.1.1 Number of releases

One of the most obvious metrics to judge if a product is well-maintained is its number of releases. In the corpus, the median number of releases per product is 3, the mean is 8.2, and the standard deviation is 26.0. Figure 5.1 displays the distribution with a cumulative curve (the x-axis is in logarithmic scale, same as other figures in the rest of this section). There are 26% of the products with only one release, 51% have no more than 3 releases, and 80% have no more than 10 releases.

Figure 5.1: The distribution of the number of releases per product in the corpus



5.1.2 Number of Identifiers

Similarly, I computed the distribution of the number of identifiers in the products. Measuring the number of identifiers can be roughly used for measuring how much

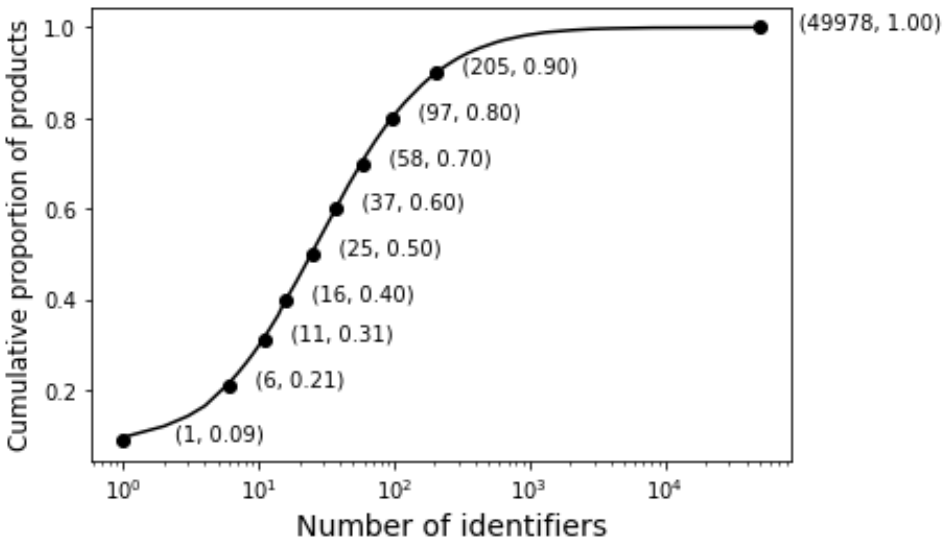
[2]As a matter of fact, the PyPI website does not provide filter/sort options like the number of releases, such that trivial products might frequently appear in searches

Python code a product has.

A product with no identifiers indicates that it is either empty or wholly written in other programming languages but distributed by PyPI for various reasons. Particularly, `XStatic` is a packaging standard (also a Python library)[3] that creates Python packages of static files (including source files written in other languages), which can be distributed by PyPI and be delivered via package managers such as PIP. For example, `XStatic-Angular` is "a Angular JavaScript library packaged for setuptools (easy_install) / pip"[4]. I identified at least 138 products in the corpus that are the XStatic packages (by naive name matching), of which 93 products have no Python identifiers. Another typical case of developers wrapping source code of other languages into products without Python identifiers was `portio`[5], which is a product for calling C library functions using a Python front end.

I counted the number of different identifier names in every product in the corpus, then found the median number of identifiers per product is 25, the mean is 113.6, and the standard deviation is 588.9. Figure 5.2 shows the distribution for these values. In total, 13,739 (5.6%) of the products have no identifiers (including the 12 products that did not provide source packages). From the figure, 9% of the products have only one identifier.

Figure 5.2: The distribution of the number of identifiers per product in the corpus



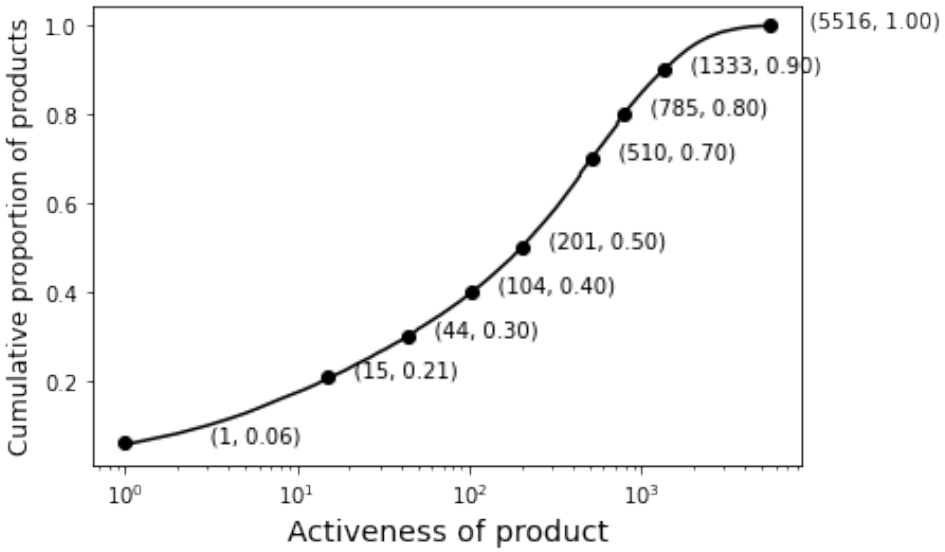
[3] `https://pypi.org/project/XStatic/`, last accessed 2022-11-01
[4] `https://pypi.org/project/XStatic-Angular/`, last accessed 2022-11-01
[5] `https://pypi.org/project/portio/`, last accessed 2022-11-01

5.1.3 Activeness of Product

I defined the *activeness* of a product as the interval between the product's first release and the latest release in the number of days. Thus, it indicates how long the product has been actively developed. A particular case is that activeness equals 0 when a product has only one release.

For 143,205 (63.8%) products with more than one release, the median of the activeness is 201 days, the mean is 472.6 days, and the standard deviation is 678.6 days. Figure 5.3 displays the distribution of the activeness for these products. The highest activeness in the corpus is 5516 days (15 years as of August 20, 2022, when the corpus was created), which is for product `ll-xist` that was first developed in 2003. The results show that the activeness for 21% of the products is no more than 15 days.

Figure 5.3: The distribution of activeness for products with more than one release in the corpus



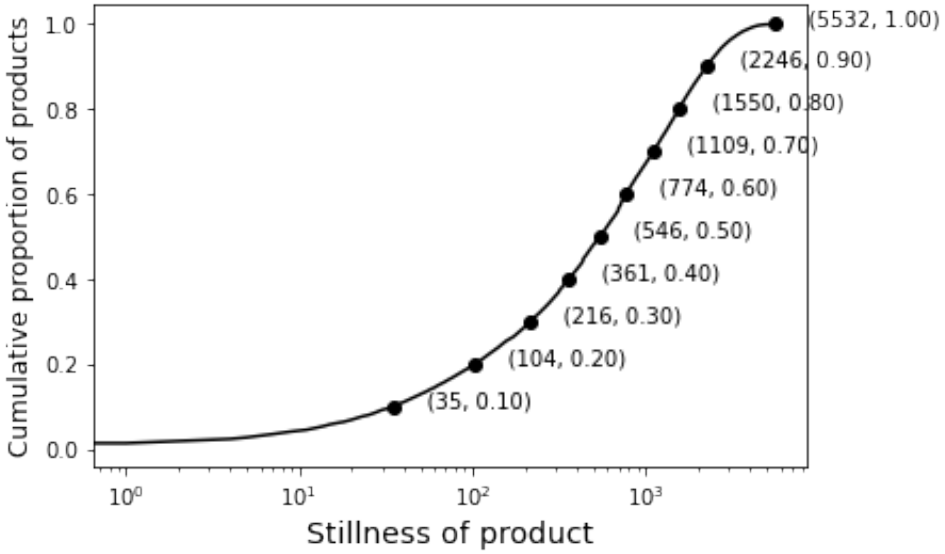
5.1.4 Stillness of Product

Newly published products have low activeness by definition; however, they might not be trivial products. So I also defined the *stillness* of a product, which is measured by the number of days from the product's last release to August 20, 2020. Stillness can be used to determine if a product is still being maintained.

I found that many products in the corpus are pretty inactive, as the median of

stillness is 546 days, the mean is 874.0 days, and the standard deviation is 931.8 days. Figure 5.4 displays the distribution, showing that the latest release for only 10% of the products was published in the past 35 days or less, and 60% (100% - 40%) of the products had no being updated for nearly more than a year (361 days).

Figure 5.4: The distribution of stillness for products in the corpus



5.2 Distribution of Distinctiveness

This section primarily describes how I empirically analyzed the distinctiveness of the identifiers in the PyPI corpus to answer the first research question restated below.

RQ 1: How unique are the class and function names among open-source Python ecosystem?

I considered the answer to the question in such aspect: *at the product level, what is the overall distribution of distinctiveness in the PyPI corpus, especially the proportion of unique identifiers (defined in only one product)?*

The following subsections report studies performed to answer this question and other related studies that provided practical insights beyond the distribution of distinctiveness.

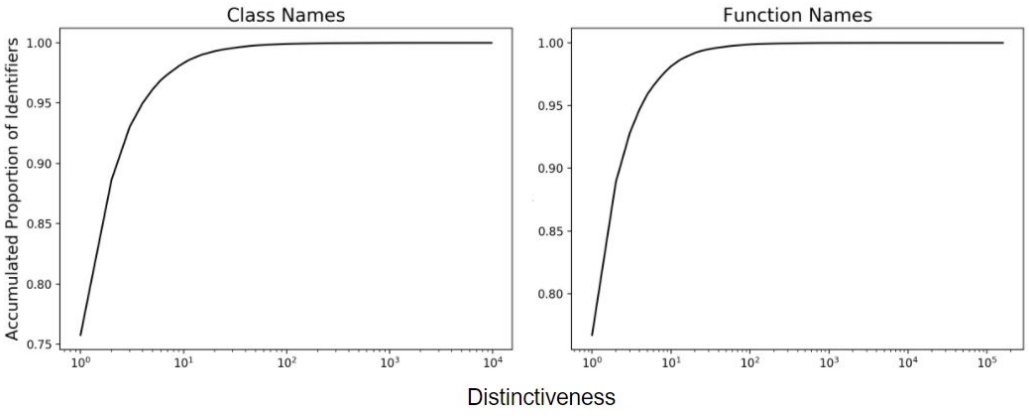
5.2.1 Proportion of unique identifiers

I calculated the distribution of distinctiveness in the corpus: for each identifier type (class and function), I obtained $dist(id, C_P)$ (i.e., distinctiveness at the product level for every identifier id); then, I counted each distinctiveness and computed its proportion to the total number of identifiers. These proportions were accumulated in ascending order of distinctiveness to report the results. Table 5.1 documents the basic and cumulative proportions of distinctiveness for each type of identifier, and figure 5.5 plots the cumulative proportions (x-axis in logarithmic scale).

Table 5.1: Distribution of distinctiveness at the product level - with respect to the number of identifiers

	Class names			Function names		
dist	# Ids	Prop. Ids (%)	Cum. Prop. Ids (%)	# Ids	Prop Ids (%)	Cum. Prop Ids (%)
1	2,020,027	75.8	75.8	6,595,770	76.7	76.7
2	343,195	12.8	88.6	1,048,985	12.2	88.9
3	117,016	4.4	93.0	335,572	3.9	92.8
4	52,326	2.0	95.0	164,056	1.9	94.7
5	30,224	1.1	96.1	101,079	1.2	95.9
6	20,528	0.8	96.9	60,735	0.7	96.6
7	12,829	0.5	97.4	44,863	0.5	97.1
8	9,786	0.3	97.7	35,849	0.4	97.5
9	8,622	0.4	98.1	28,282	0.2	97.9
10	6,623	0.3	98.3	22,702	0.2	98.1
11-100	42,241	1.6	99.9	150,015	1.7	99.9
101-1000	2,385	0.1	100	10,308	0.1	100
1001-	125	0.0	100	763	0.0	100

Figure 5.5: Cumulative distribution of distinctiveness at product level



The results show that the distributions of distinctiveness for classes and functions are very similar by comparing the proportions. In both cases, the cumulative proportions grow fast from $dist = 1$ to $dist = 3$, then the changes become minor since identifiers with $dist >= 3$ only accounted for 7%.

RQ 1 can be answered by the following observations:

Answer to RQ 1: 75.8% of class names and 76.7% of function names are unique at the product level. Moreover, 88.6%/88.9% of class/function names are defined in at most 2 products ($dist <= 2$), and 93.0%/92.8% are defined in at most 3 products ($dist <= 3$).

We can genuinely exploit this property of identifiers to develop a method for origin determination using a few identifiers.

5.2.2 Frequent identifiers

From 5.1, it is notable that 125 class names and 763 function names are defined in more than 1000 products. For each type of identifier, Table 5.2 documents the top-10 of these non-distinct (frequently seen) identifiers by their names, the number of products where they are declared, and the proportion to the total number of products.

Table 5.2: Most frequent identifier names in functions and classes

Classes			Functions		
Name	# Prs	Prop. Prs (%)	Name	# Prs	Prop. Prs (%)
Meta	9,771	4.0	__init__	159,528	65.3
Command	6,246	2.6	main	53,368	21.9
Config	6,113	2.5	run	46,158	18.9
Migration	6,062	2.5	__str__	44,664	18.3
Client	5,520	2.3	__repr__	41,796	17.1
PostInstallCommand	5,100	2.1	get	33,024	13.5
PostDevelopCommand	4,737	1.9	__call__	31,053	12.7
EggInfoCommand	4,598	1.9	setUp	30,205	12.3
User	4,291	1.8	read	25,981	10.6
Error	4,094	1.7	__getitem__	25,714	10.5

For class names, the most frequent identifiers are often related to installing and configuring a product, which is required by Python's distribution tools such as `setuptools`. In the most frequent function names, identifiers surrounded by double underscores are the built-in class method names in Python; developers can define or overwrite such methods to change the run-time behaviour of classes (e.g., `__init__` is the constructor method that is called when an object is created from a class).

For the proposed origin determination method, it is not very useful to match with these frequent identifiers since each could return a large set of candidates. Therefore, it is possible and necessary to create a list of the most frequent identifiers that should be excluded from a subject's identifier fingerprint created by the sampling algorithm (i.e., regard these identifiers as stop words).

5.2.3 Identifier names in both class and function types

Another essential characteristic of the identifiers is that only 65,982 identifier names (0.59% in total 11.2 M identifiers) in the corpus appear in both classes and functions (called *combined* identifiers). This result implies that, in the rest 99.41% of the cases, it is not necessary to keep the type of identifiers since their names tell already.

5.2.4 Probability of sampling a unique identifier

We have seen that more than 75% of the identifier names are unique at the product level, while there also exists a lot of frequent identifiers that are defined in many products. Hence, it is important to answer that if we randomly choose an identifier from the corpus, what is the probability that the identifier is unique? Note that less distinct identifiers are more likely to be sampled in this case (i.e., the distinctiveness of an identifier is the weight/bias of sampling the identifier), and the results might reflect the probability of seeing a unique identifier from the "wild" (i.e., a Python open-source ecosystem). To do that, I performed an experiment refined from Section 5.2.1: instead of finding the proportion of each distinctiveness with respect to the total number of identifiers, I calculated the proportion with respect to the number of *identifier instances* in the collection of products. An identifier instance refers to a pair <identifier name, product>, and there exists 26.8 M such instances in total.

Table 5.3 documents the distribution of distinctiveness at product level with respect to the number of instances. For the results, I disregarded the type of identifier since it was found that we can know an identifier's type by its name in 99.4% of the cases, and the number of instances for each distinctiveness was calculated by multiplying the distinctiveness (i.e., number of products) and the number of identifiers having this distinctiveness.

Table 5.3: Distribution of distinctiveness at the product level - with respect to the number of instances

dist	# Ids	# Ins	Prop. Ins (%)	Cum. Prop. Ins (%)
1	8,600,252	8,600,252	32.1	32.1
2	1,390,971	2,781,942	10.4	42.5
3	452,285	1,356,855	5.1	47.6
4	216,279	865,116	3.2	50.8
5	131,248	656,240	2.5	53.3
6	81,232	487,392	1.8	55.1
7	57,686	403,802	1.5	56.6
8	45,620	364,960	1.4	58.0
9	36,885	331,965	1.2	59.2
10	29,308	293,080	1.1	60.3
11-100	192,252	4,682,191	17.5	77.8
101-1000	12,695	3,030,540	11.3	89.1
1001-	888	2,913,499	10.9	100

The results show that 32.1% of the instances are unique identifiers, and the distinctiveness is no more than 4 in 50.8% of the instances. Hence, RQ 1 can be discussed further as:

Further answer to RQ 1: If an identifier is randomly picked, the probability of being defined in one product is 32.1%, and it is defined in no more than 4 products with a probability of 50%.

5.2.5 Distinctiveness of filenames

In Python, base filenames (without package paths and the `.py` extension) correspond to module names (e.g., file `foo.py` can be imported with `import foo`). As an extension of the study, it was interested to discover whether filenames in Python have similar distinctiveness properties as identifiers does. In this way, we might not need source code but only filenames to determine the origin, as long as these files have been copied without name changes.

Therefore, I analyzed the distinctiveness of filenames at the product level. For

this purpose, I retrieved the filenames from the collection of files in the corpus, and repeated the method in Section 4.1.4 to build another collection of products and their filenames. In total, I retrieved 1,117,588 different filenames.

Then, I repeated the experiments done in this section for calculating the distribution of distinctiveness of filenames, with respect to both the number of filenames and the number of $(filename, product)$ instances. Table 5.4 documents the results, which shows that filenames are slightly more distinct than identifiers at the product level, as 79.3% of the filenames are unique (compared with about 76% for identifiers), and 90.3% of the filenames are in no more than 2 products (about 89% for identifiers). Also, for a random filename sampled in the corpus (sampling biased by distinctiveness), in 30% of the cases it is unique, and in 42.7% of the cases it has a distinctiveness of no more than 3.

Table 5.4: Distribution of distinctiveness of filenames at the product level

dist	# Filenames	Prop. names (%)	Cum. Prop. names (%)	# Ins	Prop. Ins	Cum. Prop. Ins
1	886,236	79.3	79.3	886,236	30.0	30.0
2	122,929	11.0	90.3	245,858	8.3	38.3
3	43,064	3.9	94.2	129,192	4.4	42.7
4	17,234	1.5	95.7	68,936	2.3	45.1
5	10,038	0.9	96.6	50,190	1.7	46.8
6	6,458	0.6	97.2	38,748	1.3	48.1
7	4,638	0.4	97.6	32,466	1.1	49.2
8	3,273	0.3	97.9	26,184	0.9	50.1
9	3,004	0.3	98.1	27,036	0.9	51.0
10	2,886	0.3	98.4	28,860	1.0	51.9
11-100	15,885	1.4	99.8	421,320	14.3	66.2
101-1000	1,795	0.2	100	453,591	15.4	81.6
1001-	148	0.0	100	543,811	18.4	100

5.3 Using identifiers to determine the origin of a source code artifact

In the previous section, we found that many identifiers are very distinct at the product level. This fact created a promising building block that identifiers are useful for a lightweight method to determine the origin of Python source code artifacts within an open-source ecosystem like PyPI. However, it is important to measure how *efficient* the method is, such that the origin or a satisfyingly small number of candidates can be returned by using only a small number of identifiers. The question corresponds to RQ 2:

RQ 2: How many identifiers are needed to *narrow down* the origin of a given source code artifact to a relatively small set of candidates (ideally one) when searching within the Python open-source ecosystem?

We can again focus on the product level origin determination, then the research question is: *At product level, how many identifier names from a subject entity are required to reduce the candidate set to a certain small size (e.g., 1, 2, 3...)?* This way, we can determine an ideal number of identifier names, with which the number of candidates can be reduced to a marginal value (i.e., a point of diminishing returns, from which using more identifiers merely improves the performance).

I created a series of experiments to find the answer and turn the proposed method into a practical heuristic. For a subject entity (one or more files) sampled from the PyPI corpus, I used different numbers of identifier names (regardless of the type of identifier) sampled from the subject to create a fingerprint, matched it to the collection of products and recorded the number of candidates returned. I repeated this process until I obtained enough records. Also, I considered using a *blocklist* of identifiers, in which identifiers are too frequent to be helpful, to avoid identifiers being included by the fingerprints. Lastly, I measured the time efficiency of the implemented sampling and matching methods. Details of the experiments are presented in the following subsections.

5.3.1 Experiment Settings

I used subject entities from the latest releases of products so that the subjects were likely from different products. Each experiment comprises a sampling phase and a matching phase. First, I sampled N (1,000) latest releases, then sampled M different identifiers ($M \in [1, ..., 5]$) from each of the releases to form a fingerprint, using both of the single-file and disjoint-file sampling method (as described in Section 4.2) so that each subject can be either one or a set of M files. As mentioned, I applied a blocklist of K most frequent identifiers (called a *K-blocklist*) at the product level so that the fingerprints would not contain these identifiers.

I recorded the number of candidates returned for each M in the matching phase using the algorithm. I also timed how long it takes each phase run (experiment machine specifications: Intel i7-4790 CPU, 8 GB RAM, 500 GB HDD; database tables use optimal indexes) in order measure the time efficiency.

Sampling phase

In detail, Algorithm 1 explains the procedures for sampling the subjects and identifiers for the single-file and disjoint-file methods. Note that the experimental subjects could not be sampled from a significant portion of the latest releases as they do not have 5 different identifiers, excluding those in the blocklist. For example, only 83.9% (out of 244,084 products) of the latest releases have at least 5 different identifiers that are not in the blocklist of 300 most frequent identifiers. In the algorithm, if a sampled release does not satisfy the sampling requirements, it will be skipped, and another release will be sampled until sampling 1,000 releases.

Algorithm 1 Sampling subject entities from N latest releases, and M identifiers (that are not in the K-blocklist) from each release

Input: C_{RL}: collection of latest releases; N, M, K: integers; ST: identifier sampling method, BL: K-blocklist at the product level

Output: *samples*: a list of fingerprints (sets of sampled identifiers)

Result: experiment samples of M identifiers' fingerprint from each of the N latest releases

```
samples ← empty list
while |S| < N do
    r ← release sampled from C_RL with replacement
    if ST is single-file then
        F ← set of files in r with at least M different identifiers that are not in BL
        if F ≠ ∅ then
            f ← file sampled from F
            fingerprint ← set of M identifiers sampled from Defs(f) \ BL
            samples.append(fingerprint)
        end if
    else if ST is single-file AND r has at least M files then
        fingerprint ← ∅
        F ← set of files in r
        while F ≠ ∅ AND |fingerprint| < M do
            f ← file sampled from F
            new_ids ← Defs(f) \ fingerprint \ BL
            if new_ids ≠ ∅ then
                id ← identifier sampled from new_ids
                fingerprint.add(id)
                if |fingerprint| = M then
                    samples.append(fingerprint)
                    break
                end if
            else
                F.remove(f)
            end if
        end while
    end if
end while
return samples
```

Matching phase

Each sampled fingerprint of M identifiers was used in the matching phase to find the candidate products. By steps, I increased the number of identifiers (denoted by m) to use, from 1 to M, for finding the relation between the number of identifiers and the size of the candidate set.

Importantly, for experiment variations, I defined two matching methods for searching products that have defined a set of identifiers, as follows:

via-$\mathbf{C_P}$: Directly match against C_P to obtain a set of products that have defined all of a given set of identifiers. This method is applicable for both sampling methods, single-file and disjoint-file.

via-$\mathbf{C_F}$: Intermediately match against C_F to obtain files that have defined all of a given set of identifiers, then obtain a set of products containing each of these files. This method is only applicable to the sampling method single-file.

The matching is performed in the following way, for the sake of computational efficiency: given a list M identifiers with a random order (converted from a fingerprint), for matching with 1 identifier ($m = 1$), I use the first identifier in the list, and maintain the set of matched entities (products or files, depending on the matching method chosen); then, for matching with 2 identifiers ($m = 2$), I obtain the set of entities matched with the second identifier and compute the intersection with the previous set... This process repeats until the M-th identifier has been tested. In other words, every tested set of m identifiers contains the previously used $m-1$ identifier(s). This approach guarantees that the number of candidates does not increase as more identifiers are used.

Algorithm 2 describes the procedures for matching, respectively, for the two matching methods. The output is a raw record for statistical analyses: a mapping from the number of identifiers to a list of number of candidates that have ever been found.

Algorithm 2 Matching the samples to obtain the number of candidates by using each number of identifiers (from 1 to M)

Input: C_P: collection of products; C_F: collection of files; *samples*: list of fingerprints (sets of sampled identifiers); M: maximum fingerprint size, MM: matching method

Output: *records*: map of the number of identifiers used to lists of the numbers of products returned

Result: experiment results as described by *records*

1: $records \leftarrow \{m$: empty list **for each** $m \in 1$ to $M\}$
2: **for each** $fingerprint \in samples$ **do**
3: $ids \leftarrow$ a list of identifiers in random order from $fingerprint$
4: **if** MM is $via - C_P$ **then**
5: $candidates \leftarrow \varnothing$
6: **for each** $m \in 1$ to M **do**
7: $id \leftarrow ids[m-1]$
8: $matches \leftarrow \{p | p \in C_P \wedge id \in Def(p)\}$ ▷ the set of products defined id
9: **if** $candidates = \varnothing$ **then**
10: $candidates \leftarrow matches$
11: **else**
12: $candidates \leftarrow candidates \cap matches$
13: **end if**
14: $records[m].append(|candidates|)$
15: **end for**
16: **else if** MM is $via - C_F$ **then**
17: $files \leftarrow \varnothing$
18: **for each** $m \in 1$ to M **do**
19: $id \leftarrow ids[m-1]$
20: $matches \leftarrow \{f | f \in C_F \wedge id \in Def(f)\}$ ▷ the set of files defined id
21: **if** $files = \varnothing$ **then**
22: $files \leftarrow matches$
23: **else**
24: $files \leftarrow files \cap matches$
25: **end if**
26: $candidates \leftarrow \{$product of f **for each** $f \in files\}$
27: $records[m].append(|candidates|)$
28: **end for**
29: **end if**
30: **end for**
31: **return** *records*

Experiment Methods

I combined the sampling method and the matching method in three ways that are specified as three experiment methods as below:

- **Method A**: single-file sampling and via-C_F matching
- **Method B**: single-file sampling and via-C_P matching
- **Method C**: disjoint-file sampling and via-C_P matching

I conjectured that the results from Method C would generally produce the lowest number of candidates over the other methods. The reason is that single files might appear in different products, and using a set of files (one identifier from each) tend to locate a product more precisely. Also, Method A should work better than Method B due to the additional constraint in the matching: the identifiers should be present in the same product and the same file. Regarding time efficiency in the matching phase, via-C_F should be slightly slower than the other method since this method relies on more relations (e.g., finding the product of a file) in the database.

5.3.2 Optimal blocklist size

Applying a blocklist in the sampling phase could refine the matching results since using a set of less frequent identifiers obviously leads to smaller candidate sets. However, if the size of the blocklist (K) becomes too large, it becomes hard to sample enough identifiers as fingerprints for most subjects. Therefore, it is crucial to determine a blocklist size K that can be used throughout the experiment since this value would significantly impact the results.

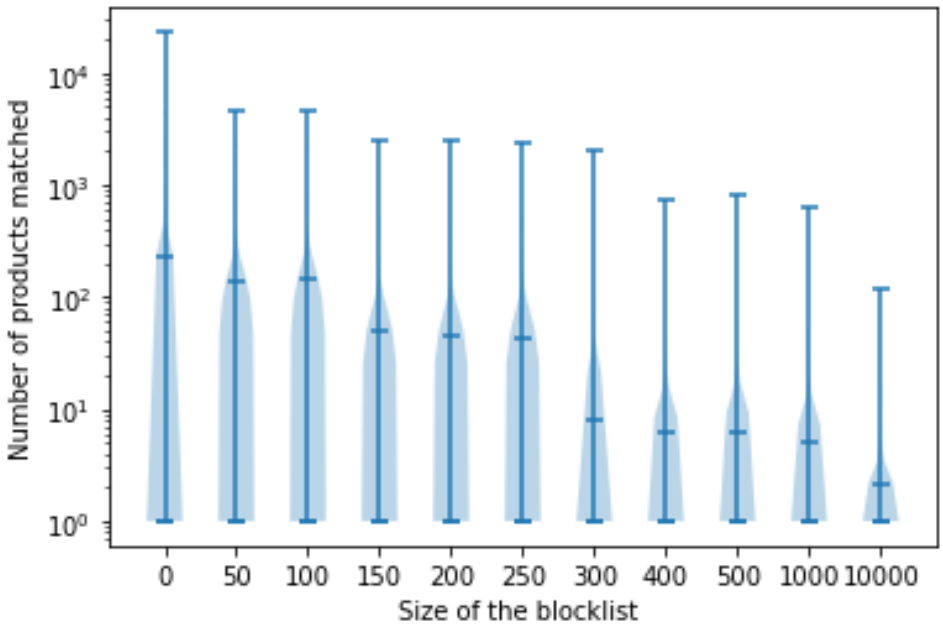
I designed an experiment to find the optimal K, such that, further increments will no longer decrease the number of candidate products in the experiment by a significant level. As described previously, the test was operated by the sampling-matching approach with Method B. Yet, I applied some modifications to the algorithms in the following way: I set the parameters as $N = 5,000$ and $M = 3$ while K is the only variable. I used a list of discrete values $K \in$ [0, 50, 100, 150, 200, 250, 300, 400, 500, 1,000, 10,000] to determine the differences and choose the optimal K within these values.

I supposed that increasing K would potentially lead to three effects that are classified as positive and negative. In the positive aspect, the first effect is decreasing

the number of candidates, as discussed. The second positive effect is shortening the matching time because smaller numbers of instances in the database will be queried. In the negative aspect, more releases will be skipped in the sampling phase since fewer identifiers are eligible; consequently, the experiment becomes more biased as these releases that are likely from trivial products are omitted. Therefore, I used three corresponding criteria to represent the influences of changing K: the number of candidates, matching time, and the number of releases examined as sampling the 5,000 fingerprints is done.

Figure 5.6 displays the number of candidates returned using each value of K. The median number of candidates is 1 for any K (not presented by the figure). However, the mean number (annotated with the middle bar in each violin plot) drops most significantly from $K = 250$ to $K = 300$.

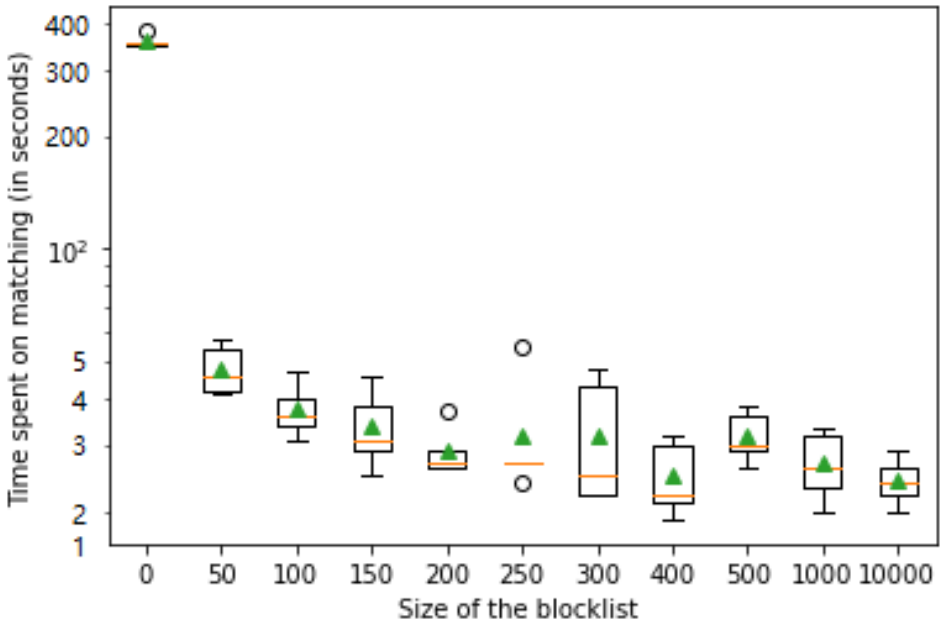
Figure 5.6: Positive effect of increasing the blocklist size: reducing the number of candidates



To evaluate the matching time, I ran the test in 5 iterations (i.e., 5 runs for each K) to allow variances. Figure 5.7 shows the results (with y-axis in logarithmic scale), in which the matching time decreases are not as consistent as I expected. I conjecture that the matching time could have been affected by various factors, such as the caching mechanisms in the database and operating system, the temporal performance of the CPU and drive, etc. What can only be observed is the matching time decreases as

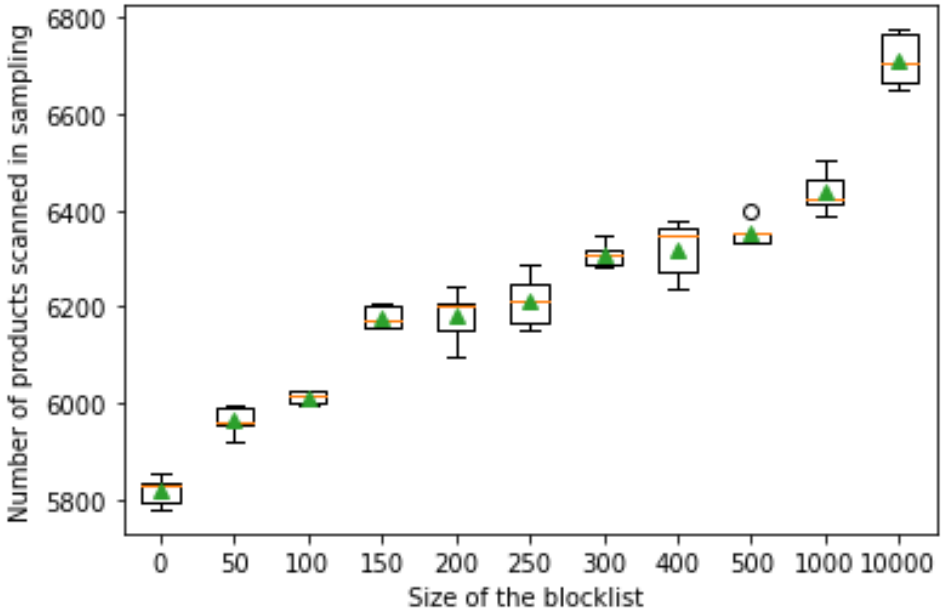
K increases within the range [0, 200], yet this fact does not continue for larger Ks. Generally, matching all the 5,000 fingerprints takes less than 10 seconds when K is larger than 50.

Figure 5.7: Positive effect of increasing blocklist size: reducing the matching time



Regarding the effect on the number of releases scanned during sampling, I also ran 5 iterations to acquire reliable results, as shown by Figure 5.8. More releases were scanned as K increased, but the change is less than 1,000 releases for expanding K from 0 to 10,000. No marginal K could be found since the sampling time rises stably as K grows.

Figure 5.8: Negative effect of increasing blocklist size: increasing the number of releases scanned for sampling 5,000 fingerprints



To conclude the results for the three criteria above, using a 300-blocklist appears to be the optimal choice for matching at the product level in the PyPI corpus only because using this size can sufficiently decrease the search space. It is also determined that the identifiers in the 300-blocklist account for 7.6% (2.0 M out of 26.8 M) of the <identifier, product> instances in C_P, which reflects the chance of encountering a blocked identifier in a Python open-source ecosystem.

5.3.3 Experimental results

I experimented respectively with each method A, B and C to answer the second research question. I let parameters other than the experiment method be fixed: N (the number of fingerprints to sample) = 1,000, M (the number of identifiers in each fingerprint) = 1 to 5, K (the size of blocklist) = 300. I ran each set for each method for 5 iterations. I also pre-computed the lists of the latest releases that are eligible for each sampling type given the parameters to avoid unnecessary scans by the sampling algorithm.

The results are reported by two criteria: the number of candidates returned, which also reflects how effective the proposed method is; the execution time (particularly in the matching phase), which reflects the method's efficiency.

Number of candidates

I will demonstrate the results regarding the number of candidates from two perspectives: the means/medians and the distribution.

For the first, I display the overall numbers of candidates for each method and each size of fingerprint m, as shown in Table 5.5. Means were computed by the **mean of the means** from results of the 5 iterations; similarly, medians were obtained from the **median of the medians**. An additional column, Δ Mean, describes the means' changes to find the marginal point.

Table 5.5: The number of candidates for using fingerprints with different sizes

	Method A			Method B			Method C		
# Ids Used (m)	Mean	Δ Mean	Med.	Mean	Δ Mean	Med.	Mean	Δ Mean	Med.
1	168.1		5	182.2		5	195.4		5
2	13.5	-92.0%	1	20.4	-88.8%	1	6.7	-96.6%	1
3	10.9	-19.3%	1	15.3	-25.0%	1	2.4	-64.2%	1
4	10.3	-5.5%	1	14.2	-7.2%	1	1.9	-20.8%	1
5	9.5	-7.8%	1	13.7	-3.5%	1	1.5	-21.1%	1

From the table, for all the methods, the numbers of candidates decline most significantly as m increases from 1 to 2, then all the medians remain at 1 product. A notable fact is that $m = 3$ is considered an inflection point because after that (between $m = 4$ and 5), the changes of the means start to be small (Method A and B) or almost zero (Method C). Hence, using 3 identifiers to determine the product-level origin is considered an optimal choice. Another observation is that Method C consistently outperform the others by having the smallest number of candidates except for $m = 1$, and the second best is Method A, which proved my conjecture: *the order of performance regarding the number of candidates is Method C > Method A > Method B.*

For the second perspective, I computed the distributions of the number of candidates for each experiment setting. For each experiment method and each m, I computed the proportion of each number of candidates out of all the records. One could use the distribution to answer questions such as "what is the chance of returning the precise origin (only one candidate) with Method A and only one identifier". I only focused on the proportions for 1 to 5 candidates because I had found that the

median values were no more than 5.

Figures 5.9 to 5.11 display the distributions of proportions for each method. The boxplots show the variances of the proportions for the 5 iterations. Also, proportions are displayed cumulatively, such that the plots describe the proportions for numbers of candidates that happened to be **less or equal to** specific values. For example, in Figure 5.10, when 3 identifiers are used, the number of candidates is no more than 3 in 89.2% (the mean from the 5 iterations) of the cases.

Figure 5.9: The distribution of the number of products matched (1 to 5) with respect to each number of identifier(s) used, Method A

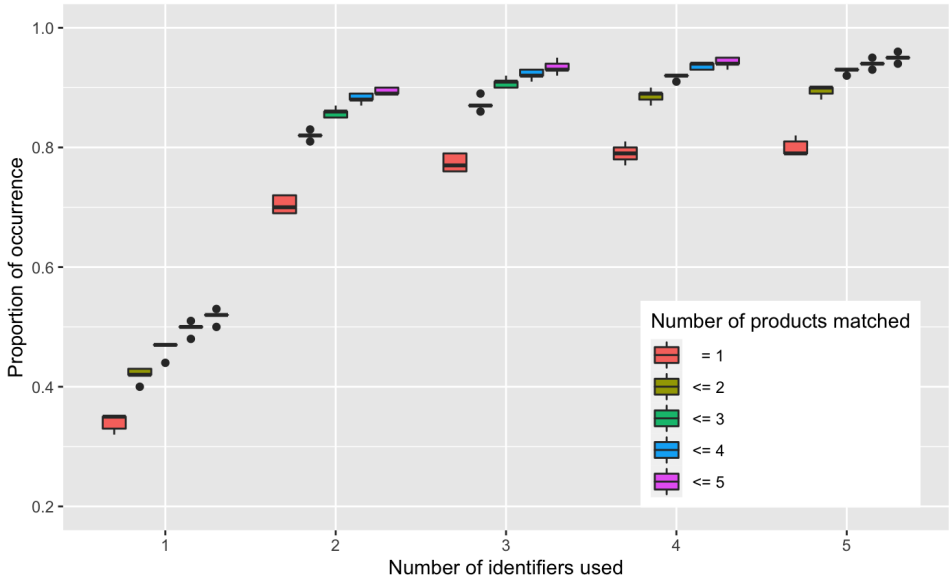


Figure 5.10: The distribution of the number of products matched (1 to 5) with respect to each number of identifier(s) used, Method B

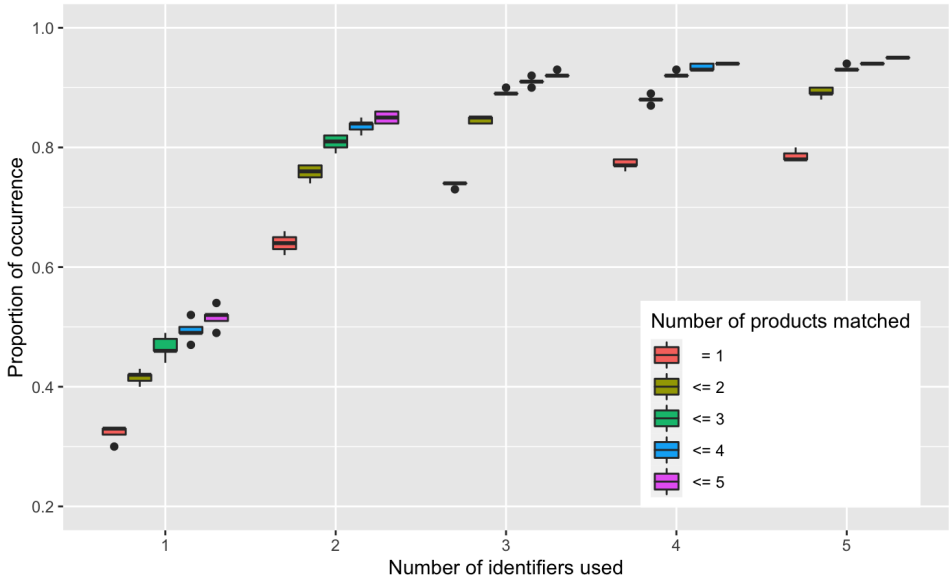
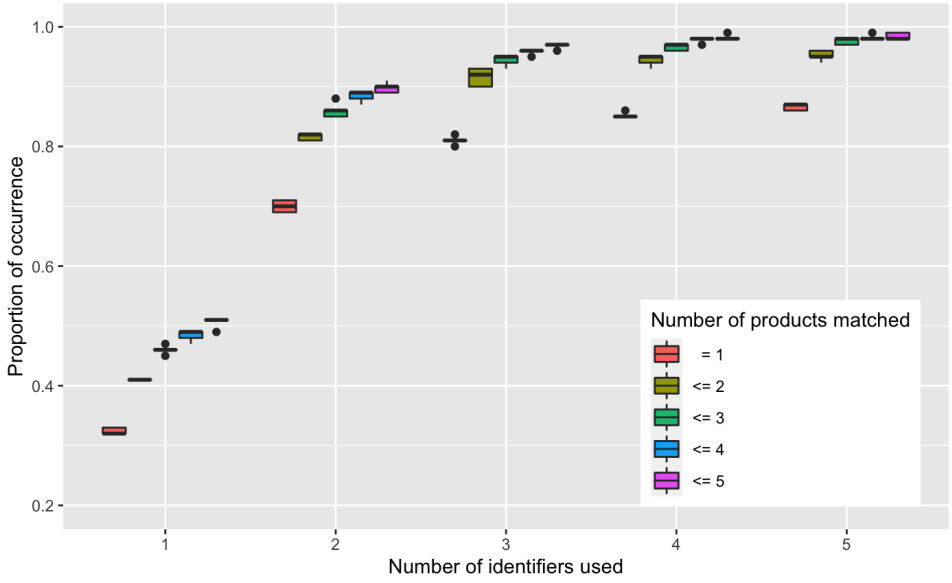


Figure 5.11: The distribution of the number of products matched (1 to 5) with respect to each number of identifier(s) used, Method C



The previous conclusions can also be verified with the results in these figures. First, overall the proportions of the low candidate set sizes from Method C are obviously higher than those from the other methods, indicating a higher chance of

returning small candidate sets. Second, using fingerprints of 3 identifiers is generally an optimal choice because these proportions remained approximately at the same level for increasing the fingerprint size from 3, in all three methods.

Execution times

I recorded the execution times of the experiments to measure efficiency. For each method, I computed the mean, median and standard deviation of the sampling and matching times from the 5 iterations, as documented in Table 5.6. The results comprise big standard deviations for some settings because the caching mechanisms made the second iteration significantly faster than the first iteration, and then the time starts to converge since the third iteration (in both sampling and matching phases). Therefore, the medians are more useful for references.

Table 5.6: Execution time of all the experiment when running 5 iterations of 1,000 times for each method (in seconds)

	Sampling			Matching			Total		
Method	Mean	Median	Stdev	Mean	Median	Stdev	Mean	Median	Stdev
A	37.5	38.5	2.4	304.7	184.7	247.6	342.2	221.3	245.7
B	53.0	52	2.8	72.0	72	6.3	125.4	125	9.1
C	50.0	38	19.0	60.8	57	12.4	111.0	95	31.3

To determine the time efficiency of the origin determination method, we would only focus on the **matching** phase. Method B and C use the same matching type $via - C_P$, so their results were close. Method A is relatively slower (the median is about 3 times as the other two's: 184.7 versus 72 for B and 57 for C) because with matching type $via - C_F$, more (identifier, file) instances were matched compared to matching (identifier, product) instances with the other matching type. Recall that the matching phase in each iteration included querying for each of the 5 identifiers in each of the 1,000 fingerprints, hence the total matching time approximately equals the aggregation of querying identifiers in the database 5,000 times (ignoring time costed on other operations). Therefore, even with the slowest matching type (in Method A), looking up one identifier only took 36.9 milliseconds (184.7 seconds /5,000) on average.

5.3.4 Conclusion of the experiments

Finally, I summarized the properties of the three experiment methods and answered the second research question.

Method C outperformed the other two methods since it generally returned the fewest candidate products; however, its strict sampling requirement caused a lot of releases to be skipped since there were few files. Yet, in actual use cases, if the subject is a set of files, it is recommended to use identifiers from disjoint files. On the other hand, for single files, Method A and B both produced satisfactorily small candidate sets, while Method A was slightly better, but it the sampling took a bit longer (yet the difference is in milliseconds).

The answers to RQ 2 are below, which are common properties of all the experiment methods.

Answer to RQ 2: Generally, for determining the origin at the product level, using 3 identifiers is the optimal choice because the number of candidates will no longer decrease significantly as more identifiers are used. At that point, the number of returned candidates is no more than 3 in about 90% of the cases.

5.4 Summary

At the beginning of this chapter, I analyzed some characteristics of the PyPI ecosystem and identified that PyPI has many trivial products for the following reasons. For the first reason, a large portion of products has few numbers of releases, as 6% have only one release and 80% have no more than 10 releases. Second, many products have few or no Python identifiers for various reasons, such as the developers using PyPI casually for non-software projects or to distribute static files or software written in other languages. 5.6% of the products have no identifiers, 9% have only one, and 50% have no more than 25. Third, for products with more than one release, 21% of products had only been active for 15 days or less. Lastly, only 10% of products were updated in the past 35 days, and 60% had been still for almost a year.

I found empirical evidence through the experiments to answer research questions 1 and 2. For RQ 1 (*How unique are the class and function names among open-source Python ecosystem?*), I computed the distributions of identifier distinctiveness at the product level. For distribution regarding the number of identifiers, about

76% of the identifiers are unique, and 89% are defined in at most 2 products. For distribution regarding the number of instances (i.e., pairs of (identifier, product)), 32.1% of the instances are unique identifiers, and 50.8% are identifiers in at most 4 products. Besides, only 0.59% of all the identifier names are from both classes and functions. For RQ 2 (*How many identifiers are needed to narrow down the origin of a given source code artifact to a relatively small set of candidates—ideally one—when searching within the Python open-source ecosystem?*), I first found the optimal number of frequent identifiers that should not be in fingerprints is 300; then, by experiments with the defined algorithm to sample and match identifiers, I determined that using a set of 3 identifiers as a fingerprint is powerful enough to make the number of candidates less or equal to 3 most of the time (about 90% with all the experiment methods). The proposed method is also fast since querying the product(s) that define a given identifier is almost instantaneous (in milliseconds).

Chapter 6

Evaluation

The previous chapter showed the potential of the proposed OSS origin determination method that takes advantage of the distinctiveness of code identifiers. In this chapter, I empirically evaluate the effectiveness of this method by experimenting with actual use cases where subjects are software packages found outside of the PyPI ecosystem, yet there exist identical (or nearly identical) product copies in the corpus. For example, in real-world scenarios, when a developer needs to know about the origin of a file that has been cloned or copied, it has also potentially been changed to integrate with the current software system, such that classes or functions inside have been added or deleted.

To this end, I consider a range of Python packages shipped by Debian 10 "Buster" GNU/Linux distribution, which have also been released on PyPI, as the experiment subjects. Debian ships selective packages that are in stable releases and are uploaded by official developers[1], while PyPI is considered the most comprehensive ecosystem of Python products. Hence, I assume PyPI (set of products) is the superset of the Debian packages developed in Python; then, if we know, by solid evidence, the corresponding product for each Debian package, we can use the Debian packages (called a "golden" set) with a ground truth for evaluation. We can evaluate the proposed method by identifying the origin of each package at the product level in PyPI and comparing it to this ground truth. The results can be used for answering the last research question:

> **RQ 3**: Is the proposed approach useful when the subject artifacts are *outside* the reference corpus (i.e., the PyPI ecosystem)?

[1] `https://www.debian.org/doc/manuals/maint-guide/upload.en.html`, last accessed 2022-11-07

So far, I have considered the method's effectiveness by the number of candidates returned. In this evaluation, I will be using SourceRank to refine the result (i.e., select the product that is most likely the origin).

This chapter contains the following, which will be sequentially described in the following sections.

1. I retrieved the source code of a set of selected Debian packages, then built a small Debian corpus which the experimental subjects can be sampled from. I also determined these packages' product names in PyPI as a ground truth.

2. I applied the proposed method to determine the origin of these subjects at the product level in the PyPI corpus, then compared the results to the ground truth.

3. I designed and applied a more exhaustive method that computes the similarity between the subject and every potential candidate, using all the identifiers from each side to address the false negatives (i.e., no matches) that occurred in the experiment.

4. I applied a simpler approach that only uses filenames to determine a baseline for comparison of the proposed method.

6.1 Experiment Subjects

As official Debian maintainers who oversee the packages have familiarity and expertise in their work, the metadata related to the packages, including their names and versions, is accessible and trustworthy. In this case, it is promising to obtain a golden set of Debian packages and a solid truth of origin information that I can use to validate matching results for evaluating the method.

6.1.1 Subject source code and ground truth retrieval

I retrieved data for evaluation in August 2020 (right after I created the PyPI corpus). I created a Debian 10 environment by pulling and running the official Docker image. I retrieved a complete list of Debian packages from Debian's default package manager, APT, in the `Main` section of the Debian archive[2].

[2]The Main section contains the most and free packages, meanwhile relatively fewer packages exist in two other sections, namely Contrib and Non-Free. `https://www.debian.org/distrib/packages`, last accessed 2022-11-07

Like PyPI, Debian provides both downloadable source and binary packages; also, one source package might correspond to multiple binary packages, and I would choose the source version of each package. I obtained a list of all the packages in the source package index file created by APT[3]. This file contains the meta information on all the source packages for over 28,000 packages. The sample fragment below is part of the file, indicating source package `zzzeeksphinx` versioned `1.0.20-2` corresponds to two binary packages, `python-zzzeeksphinx` and `python3-zzzeeksphinx`.

```
...
Package: zzzeeksphinx
Binary: python-zzzeeksphinx, python3-zzzeeksphinx
Version: 1.0.20-2
Maintainer: Debian Python Modules Team <python-modules-team@lists.alioth.debian.org>
Uploaders: Piotr Ozarowski <piotr@debian.org>
Build-Depends: debhelper (>= 9), dh-python, python-all, python3-all, python-setuptools,
python3-setuptools
Architecture: all
Standards-Version: 4.0.0
Format: 3.0 (quilt)
Files:
 16f5bd47dd1d3d1b1095becc8d4c10dc 2145 zzzeeksphinx_1.0.20-2.dsc
 471c4b42495e76a7c23678e2f8eb2569 23030 zzzeeksphinx_1.0.20.orig.tar.gz
 32934345771155ae79fe4a19bbcc7267 5364 zzzeeksphinx_1.0.20-2.debian.tar.xz
Vcs-Browser: https://salsa.debian.org/python-team/modules/zzzeeksphinx
Vcs-Git: https://salsa.debian.org/python-team/modules/zzzeeksphinx.git
Checksums-Sha256:
...
Homepage: https://bitbucket.org/zzzeek/zzzeeksphinx
Package-List:
 python-zzzeeksphinx deb python optional arch=all
 python3-zzzeeksphinx deb python optional arch=all
Directory: pool/main/z/zzzeeksphinx
Priority: optional
Section: misc
```

Because Debian APT does not provide the language information in the metadata, I used a coarse heuristic to identify packages in Python: I selected a list of source packages that had any python-related keywords in the source package names or the corresponding binary package names (any of `py`, `py2`, `py3`, `python`, `python2`, `python3` that appears as a token[4] of a name). This way, I parsed the package index file to identify 3,155 source packages, likely Python ones. Then, I downloaded the source packages with the command `apt-get source <source package name>`; by default, APT downloaded the latest version of each package. All the compressed packages took 26 GB of disk space in total.

Through multiple steps, I narrowed the experiment subject list by filtering out packages with unclear origins in PyPI or without Python identifiers.

[3]Full filepath in Debian 10: /var/lib/apt/lists/deb.debian.org_debian_verb_dists_buster_main_source_Sources.lz4. Note that, the file was not created in Debian by default. The way to set up the file is instructed by: `https://wiki.debian.org/SourcesList`, last accessed 2022-11-07.

[4]Tokens in a name string was considered substrings split by dashes or underscores

The source package names might not match to names of related products in PyPI. Commonly, Debian maintainers rename a package with minor modifications (e.g., capitalization, separator, prefix, etc.) when the package is uploaded the first time to satisfy Debian's naming idiom. For example, PyPI product `Django` has been renamed to `python-django` in Debian. To address this issue manually, one can possibly find the product by searching for fuzzy names in PyPI, and looking up every potential product for information such as documentation, developer name(s), and project homepage. Such effort is obviously unacceptable for thousands of packages.

Instead, for an automatic approach, I narrowed down the list to packages that have the file `setup.py` in their directories. The setup file is required by `distutils` or `setuptools`, which are two libraries for PyPI product packaging[5]. A setup file contains a package's metadata, as the code block below shows a sample setup file[6]. In function `distutils.setup()` (or `setuptools.setup()`), parameter `name` declares the product name that matches to PyPI. Scanning the file contents of every Debian package, I could find the setup file in either the root directory or in `<root>/python/`. In total, 2,809 packages have setup files, so I kept them.

[5]distutils is included in the standard library and is what setuptools is based on

[6]`https://packaging.python.org/tutorials/packaging-projects/`, last accessed 2022-11-07

```
import setuptools

with open("README.md", "r", encoding="utf-8") as fh:
    long_description = fh.read()

setuptools.setup(
    name="example-pkg-YOUR-USERNAME-HERE",
    version="0.0.1",
    author="Example Author",
    author_email="author@example.com",
    description="A small example package",
    long_description=long_description,
    long_description_content_type="text/markdown",
    url="https://github.com/pypa/sampleproject",
    project_urls={
        "Bug Tracker": "https://github.com/pypa/sampleproject/issues",
    },
    classifiers=[
        "Programming Language :: Python :: 3",
        "License :: OSI Approved :: MIT License",
        "Operating System :: OS Independent",
    ],
    package_dir={"": "src"},
    packages=setuptools.find_packages(where="src"),
    python_requires=">=3.6",
)
```

The next task was to extract the name parameters from the setup files, for which I tried two best-effort approaches. First, I dynamically run the setup files with the command `python setup.py --name` and with both Python 2 & 3 interpreters (since the implementing Python versions are unknown, and there are syntactic differences between these two versions). For 2,369 packages, the executions were successful so that their product names were found, yet the rest 440 failed mostly because of missing dependencies that were required to be imported by the files. As the second approach, I statically used Regular Expression to extract the name from files, which resulted in finding 132 more names (RE failed in cases where the name parameter was assigned by another variable). At this point, I identified 2,501 Debian Python packages with credible PyPI product names. However, searching these product names, I found that 280 products did not exist in the PyPI corpus. I conjecture that they had been removed or renamed by PyPI since I also found many Debian packages had not been updated for a very long time.

I extracted identifiers from the remaining 2,221 packages to build a Debian corpus, with the approaches described in Section 4.1.2. During this process, I found 40 packages without any Python identifiers detected by Ctags, so they were removed

(including 33 `XStatic` packages as discussed in Section 5.1.2). Lastly, I removed 7 more packages whose PyPI products have no identifiers.

In addition, I noticed some packages were from the same PyPI products. For example, package `python-pysqlite1.1`, `python-sqlite`, and `python-pysqlite2` were all from the popular product `pysqlite`. There were only 13 packages (from 6 products) in such a case, so I decided to simply overlook this anomaly.

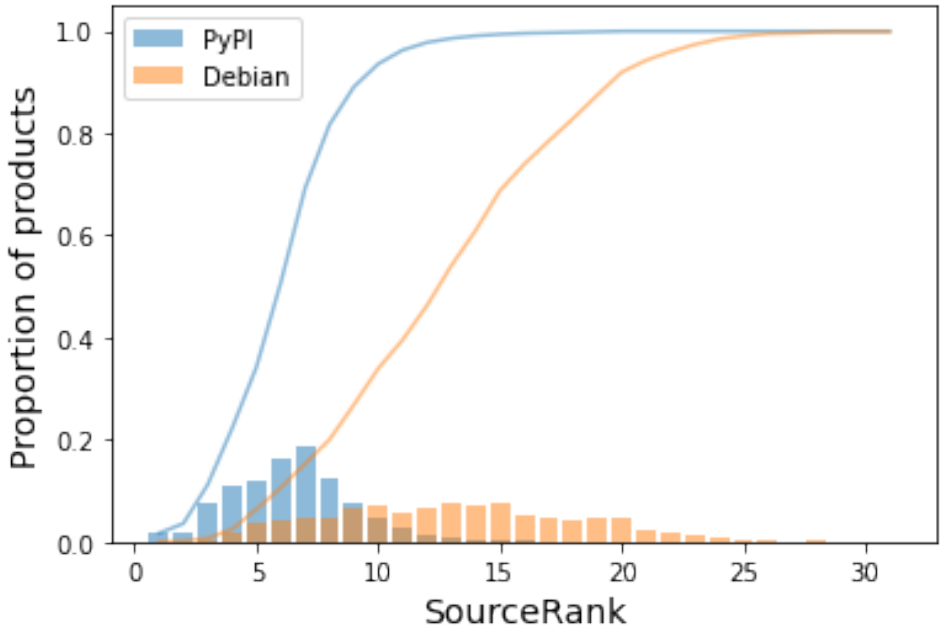
Finally, 2,174 subjects remained in my golden set of Debian packages. Although I might be able to identify more Debian packages (that are in PyPI) with more advanced methods and time/effort, the goal was not to identify as many as possible, and the current number of packages was considered sufficiently large to perform the evaluation.

6.1.2 Empirical statistics

I list some empirical statistics to demonstrate that Debian packages are high-quality, so SourceRank should perform well in sorting the candidates. I focused on the subset of PyPI products from which the 2,174 Debian packages came.

First, I plotted the distribution of the SourceRank for products in the subset and for products in the entire PyPI corpus (as presented in Section 4.1.3). Figure 6.1 shows the comparison. Clearly, the products in the subset are better ranked than the average in PyPI. In numerical values, the mean SourceRank in the subset is 13.1 and the median is 13, contrasting to PyPI with a mean of 6.5 and a median of 6.

Figure 6.1: The distributions of SourceRanks among the target products of the subjects versus among the all the PyPI products



Lastly, the products in the subset are relatively "active" and not "still" (referring to trivial metrics defined in Section 5.1) so they are considered non-trivial. The mean activeness in the subset is 2036.8 days and the median is 1949 days, while in the entire corpus, the mean is 301.6 days and the median is 18 days. The mean stillness in the subset is 723.4 days and the median is 281 days, comparing to 874.0 and 546 days for the whole corpus.

6.2 Experiment 1: 3-Identifier fingerprint method

The first experiment aimed to evaluate our basic origin determination method using identifiers as defined in Section 4.2. From each of the 2,174 subjects, I sampled a fingerprint with a fixed size of 3 and the 300-blocklist. The parameters were determined as the optimal choices based on results from the previous empirical experiments, as described in Section 5.3). Also, the evaluation focused on applying the disjoint-file sampling methods and the $via - C_P$ matching method (same as method C defined in Section sec:methods), since the package was regarded a subject as a set of files.

6.2.1 Experiment setting

Sampling phase

The heuristic of sampling identifiers to form a fingerprint for a Debian package was similar to which in Algorithm 1. As mentioned, I used $M = 3$ for fingerprint size and a 300-blocklist. However, I did not wish the sampling requirements to be as harsh as before, which caused many packages to be removed from the subjects. Hence, I made the following modifications so that, with the best effort, the origins of all the packages would be determined. First, when the total number of identifier names in a package after using the 300-blocklist was below 3, the blocklist rule was waived. Further, if there are still not enough identifiers, I used a smaller fingerprint containing all of them (i.e., a fingerprint sized 1 or 2). Algorithm 3 overall describes the refined sampling method. For evaluation purposes, I also stored the PyPI product name for each sample as the ground truth to be compared with.

Algorithm 3 Sampling a fingerprint from a Debian package subject

Input: S: subject of Debian package, M: the maximum fingerprint size, BL: 300-blocklist at the product level

Output: $fingerprint$: set of sampled identifiers

Result: at maximum M identifiers sampled from a subject

1: **if** $|Defs(s) \setminus BL| >= M$ **then**
2: $ids \leftarrow Defs(s) \setminus BL$
3: **else** ▷ There are not enough identifiers after using the blocklist
4: $ids \leftarrow Defs(s)$
5: $BL \leftarrow \varnothing$ ▷ Waive the blocklist
6: **end if**
7: $fingerprint \leftarrow \varnothing$
8: $F \leftarrow$ set of files in S
9: **while** $|fingerprint| < min(M, |ids|)$ **do**
10: $f \leftarrow$ file sampled from F
11: $new\_ids \leftarrow Defs(f) \setminus fingerprint \setminus BL$
12: **if** $new\_ids \neq \varnothing$ **then**
13: $id \leftarrow$ identifier sampled from new_ids
14: $fingerprint.add(id)$
15: **else**
16: $F.remove(f)$
17: **end if**
18: **end while**
19: **return** $fingerprint$

The relaxed sampling method helped fingerprints from a few packages to be sampled. 346 out of the 2,174 packages have only 1 or 2 files, and just 3 packages have only 1 or 2 identifiers. The mean/median number of identifiers per package is 464.9/129.5. With the 300-blocklist applied, the value drops slightly to 442.1/114, and 21 packages no longer have 3 identifiers.

Matching phase

For each subject, I matched against C_P using the fingerprint to get the candidate set; then, used SourceRanks of each product to turn the candidate set into an ordered list. I analyzed the outcomes threefold:

1. How many candidates were found?
2. If the correct origin present in the candidate set?
3. If the correct origin present at the top of the candidate list?

I adopted two metrics to answer the questions. Here, I let C denote the number of candidates returned, and R denote the rank of the *target* (i.e., correct origin product) among the candidates. There are two special cases of R: $R = 0$ indicates the target is not in the candidates (i.e., the worst case), and $R = 1$ implies the target is at the top of the candidates.

Evaluation criteria

I categorize the possible outcomes, as discussed above, as criteria of matching quality. The outcomes are within two (super) classes: **(M)atched** (success) and **(N)on-matched** (fail). Then, the outcomes are refined as subitems in the two classes, as shown below (ordered from the best to the worst case):

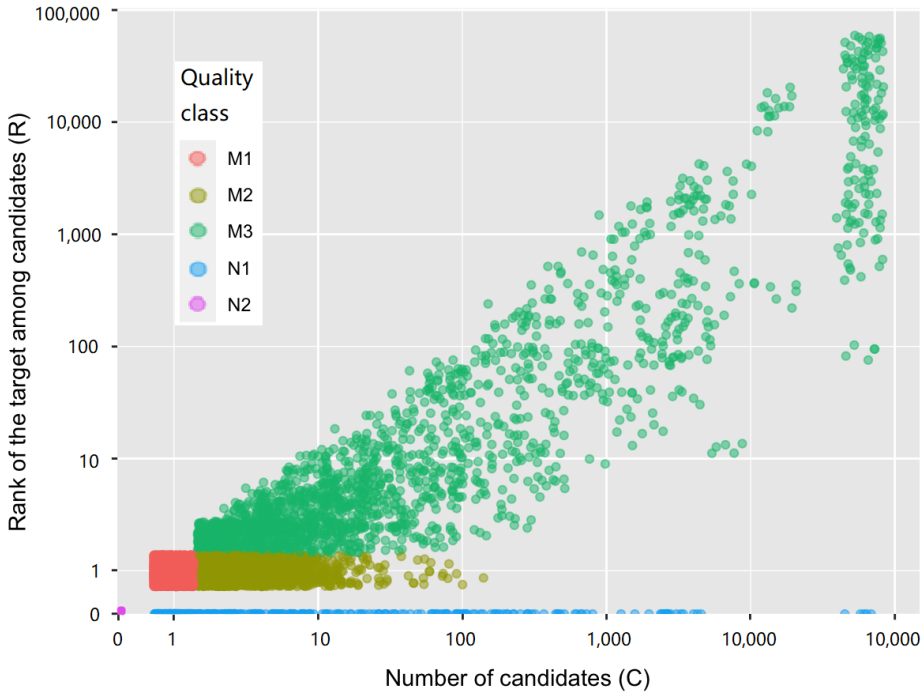
- **M1. Exact match** ($C = 1, R = 1$): there is only one candidate that is exactly the target.
- **M2. Top match** ($C > 1, R = 1$): there are multiple candidates, and the target is at the top place.
- **M3. Coarse match** ($C > 1, R > 1$): there are multiple candidates, the target is inside but not at the top place.
- **N1. Not in candidates** ($C >= 1, R = 0$): there are multiple candidates, but the target is not inside.
- **N2. No candidates** ($C = 0$): there are no candidates returned.

These items are defined as the *quality classes* in the evaluation. Because the sampling algorithm is randomized, I performed 5 trials for sampling and matching with possibly different fingerprints from each subject. The results are reported in two dimensions: *by outcomes* and *by subjects*. For the former, I listed the distributions of the quality classes for all the 2,174 $\times$ 5 outcomes. For the latter, for each quality class, I counted both 1) the number of subjects that met the quality in at least one trial and 2) the number of subjects that met the quality in all 5 trials.

6.2.2 Experiment results

Figure 6.2 demonstrates the density distribution of the returned quality classes for all the 10,870 outcomes by a density plot of C versus R (both axes are in logarithmic scales). M1 outcomes in the bottom-left corner account for the majority, followed by M2 outcomes in the area of $C < 10$, and M3 outcomes with $C < 10$ and $R = 2$ take place. There are a noticeable number of outliers for M3, as shown by the rightmost region where both C and R are high.

Figure 6.2: The number of candidates versus the rank of the target in the candidates for all the 10,870 outcomes



More detailedly, Table 6.1 summarizes the results in both the outcome and subject dimensions. Regarding the outcome dimension, in 87.1% (sum of the first three classes, 9,462 out of the 10,870 outcomes) of the cases, the targets are present in the candidate sets. I consider this value the *recall* of the experiment, and the 9,462 outcomes of successful matches are regarded *true positives*, and the 1,408 (12.9%) outcomes of failures are *false negatives*. Regarding the subject dimension, 68.1% (1,481 out of 2,174) subjects have at least once perfect M1 match, while 5.6% (121) never matched any candidates.

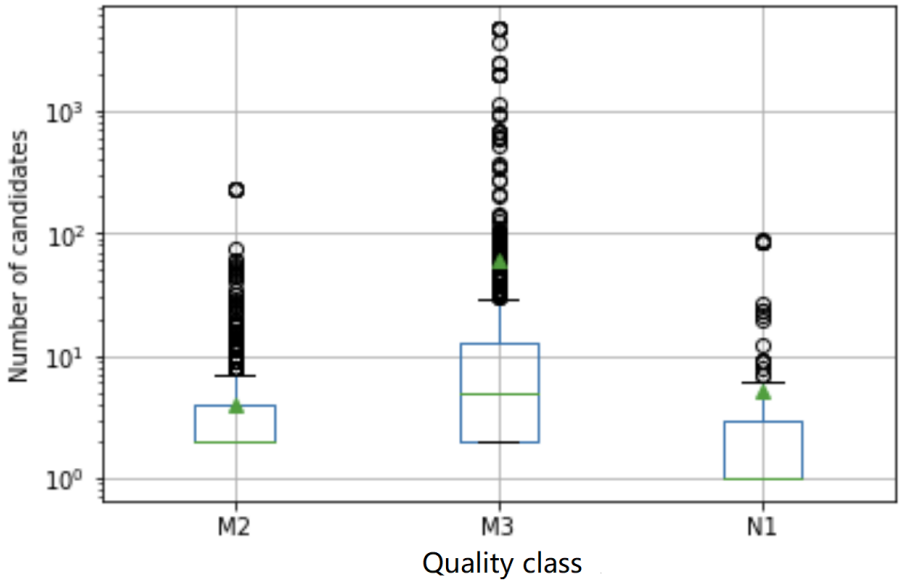
Table 6.1: The quality-categorized results in both the outcome and subject dimension - the 3-identifier fingerprint method

Quality class	# Outcomes	# Subjects, any trial	# Subjects, 5 trials
M1 ($C = 1$, $R = 1$)	5,931 (54.6%)	1,481 (68.1%)	834 (38.4%)
M2 ($C > 1$, $R = 1$)	2,395 (22.0%)	755 (34.7%)	263 (12.1%)
M3 ($C > 1$, $R > 1$)	1,136 (10.5%)	447 (20.6%)	99 (4.6%)
N1 ($C > 0$, $R = 0$)	166 (1.5%)	73 (3.3%)	15 (0.7%)
N2 ($C = 0$)	1,242 (11.4%)	381 (17.5%)	121 (5.6%)

The M1 matches account for 62.7% (M1 in M1,2,3) of the true positives, indicating that using 3 non-frequent identifiers helps find the exact match with a high probability. Slightly worse, when multiple candidates are found, the targets can still be the top ones in 67.8% (M2 in M2,3) of the times, thanks to SourceRank. Since 6.3% (15 + 121 = 136 for N1-2 in the last column) of the subjects are never found in all of the 5 iterations, I can tell the rest 93.7% (2,011) of the subjects, have been found at least once.

The boxplots in Figure 6.3 display the number of candidates in the three quality classes with multiple candidates existing (on each box, the triangle annotates the mean, and the green line annotates the median value). Particularly for M3, the median/mean number of candidates is as high as 59.3/5; yet, I have determined that the ranks of correct targets are still near the top, with a median/mean of 9.6/2.

Figure 6.3: The numbers of candidates for the outcomes in three quality classes - the 3-identifier fingerprint method



However, 6.3% of the subjects had never been matched with the method. I conjecture that there were three causes of this particular circumstance:

1. Developers put extra Python code (e.g. external dependency, usage, test, or build files), that was not part of the original product as distributed in PyPI, into these Debian packages for some reasons.

2. The Debian packages were in newer/older releases of the product that were not distributed by PyPI.

3. The way of retrieving the ground truth was unreliable, such that some package names in setup files have nothing to do with PyPI.

4. PyPI packages do not include all source files but only binary distributions.

In the next section, I will use a more exhaustive method than this 3-identifier fingerprint method to find credible matches in PyPI for the 136 subjects.

6.3 Experiment 2: Exhaustive Method

This method adopts a similarity-base heuristic such that, for **every identifier** in the subject (as a large fingerprint), find the products in the reference corpus that

defined the identifier, then return a set of candidates that contains products with the **most common identifiers**. This way, the candidate set is very likely to be a singleton, and the (only) product in the set is convincingly the target since it has the largest intersection with the subject. This method is considered more exhaustive due to the overhead for querying more than 3 identifiers. However, this method requires only one trial because it does not include a randomized sampling mechanism. (The detailed algorithm procedures are not included since the method is considered straightforward.)

To measure what proportion of identifiers in an entity with respect to those in another, I used the inclusion index defined as follows.

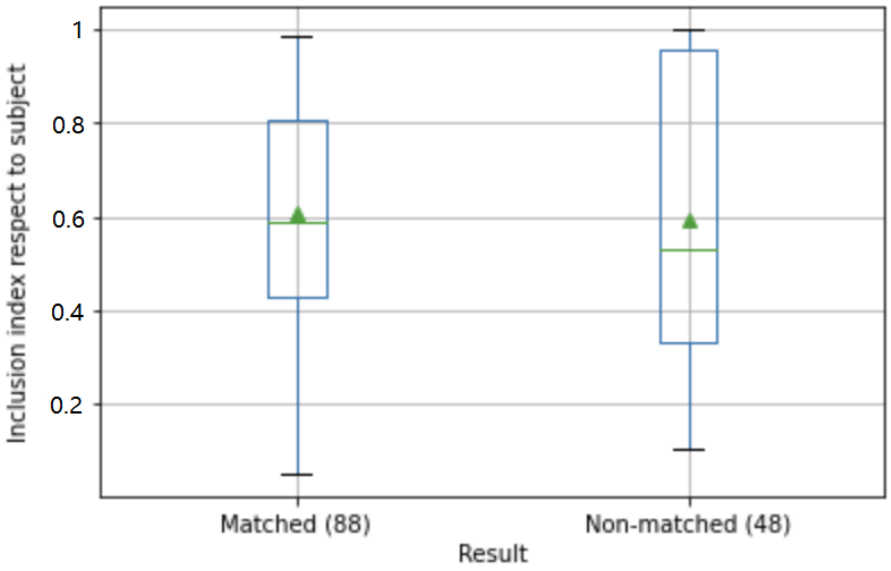
$$incl(A, B) = \frac{|Def(A) \cap Def(B)|}{|Def(A)|}$$

$incl(A, B)$ denotes the intersection of identifiers in both entity A and B with respect to the number of identifiers in A. For a special case, when inclusion equals 1, all the identifiers in A can be found in B. In our case, for example, $incl(subject, product)$ computes the proportion of identifier intersection (between the subject and a product) with respect to all the identifiers in the subject.

6.3.1 Results

I applied the exhaustive method to find the PyPI origins for the 136 subjects that produced false negatives with the 3-identifier fingerprint method. Finally, every subject is matched to at least one candidate due to the property of this method. In the results, 88 of the subjects are matched (M result), as their targets are inside the candidates (in which 75 are M1 exact matches); whereas 48 of the subjects are still non-matched (N1 result). I computed the distribution of the inclusions $incl(subject, topcandidate)$ as shown by Figure 6.4; from which, for both sets of results, the top candidates generally contain 60% (read from the means) of the identifiers in the subjects.

Figure 6.4: The distributions of the inclusion indexes with respect to the subjects via the exhaustive method



As an expanded version of the 3-identifier fingerprint method, the exhaustive method definitely could also find targets for the 2,011 subjects that were already matched to their correct products (at least once) in the previous experiment with the 3-identifier fingerprint method. With the 88 new matches, the exhaustive method increases the recall from 87.1% to 96.6% (2,099/2,174).

6.3.2 Qualitative analysis for the false negatives

Yet, it is still necessary to determine the reasons for the rest 48 non-matched results. To this end, I first analyzed identifier similarity between each subject S and 1) its ground-truth (target) product T in PyPI, as well as 2) the best-matched yet wrong candidate C (one or more). I computed $incl(S, T)$ and compared it with $incl(S, C)$, to inspect how likely the ground truth I retrieved was invalid (i.e., C is the actual origin rather than T). For T, I computed the opposite inclusion index, $incl(T, S)$, which could represent two-way similarity. Besides, I checked if the T had ever released the same version as the subject by a basic version string matching (as indicated in column "ver."). Finally, Table 6.2 documents part of the findings (with representative reasons) ordered by $incl(T, S)$ then $incl(S, T)$.

Table 6.2: Qualitative analysis of selected 34 cases for reasoning false negatives from the exhaustive method

S	$\|Def(S)\|$	T	$\|Def(T)\|$	$incl(S,T)$	$incl(T,S)$	Ver.	C	$incl(S,C)$
python-statistics	18	statistics	17	0.94	1.00	yes	avendesora	1.00
slowaes	40	slowaes	28	0.70	1.00	yes	FolsTools	0.75
pigpio	223	pigpio	155	0.70	1.00	no	auto-pi-lot	1.00
python-cymruwhois	37	cymruwhois	23	0.62	1.00	yes	dnsdiag	0.68
...								
python-boto3	1008	boto3	342	0.34	1.00	yes	dataspine-cli	0.35
...								
python-jsonpath-rw	127	jsonpath-rw	98	0.66	0.86	yes	jsonpath-ng	1.00
...								
pytest-tornado	57	pytest-tornado	21	0.30	0.81	yes	pytest-tornado5	0.98
...								
python-uritemplate	23	uritemplate	99	0.35	0.08	yes	appengine-sdk	0.43
python-dlt	180	dlt	13	0.01	0.08	no	ACOio, appengine-sdk	0.28
rainbow	68	rainbow	60	0.06	0.07	no	sickchill	0.29
xmlelements	93	xe	15	0.01	0.07	no	pydruid	0.30
python-pdftools	105	pdftools	54	0.01	0.02	no	avendesora	0.36
kiwi	1405	kiwi	3625	0.03	0.01	no	kiwi-gtk	1.00
pymia	2	pymia	407	0.00	0.00	no	xbob.extension, xbob.flandmark, xbob.optflow.liu, xpra	1.00

From the selected cases, 34 target products contain releases with the same versions as the subjects, while 14 do not. However, In 27 of these cases, the value of $incl(T,S)$ is larger than 0.8. Here I listed a few particular cases, for which I have manually inspected the metadata and source code in both the subject and the target, then classified the findings into 3 reasons:

1. **Extra code in package** and **package copied by other products**: For Debian package `python-statistics` (3.4.0b3) and `python-boto3` (1.9.86), $incl(S,T)$ is just a bit smaller than $incl(S,C)$ so that their targets were non-matched. The first reason is that both packages contain source code that is not in their targets. Identifiers in both packages are a superset of identifiers in their target products (as $|Def(S)|$ is larger than $|Def(T)|$ and $incl(T,S)$ is 1). Also, the additional identifiers are from files with test suits in both cases. It seems a common practice that Debian maintainers bundle Python software with test and build files along with the source code (likely by standard), while this occasion is less frequent in PyPI. From another perspective, the reasons why the mismatched candidates contain more identifiers than the targets do are different in the two cases. Product `avendesora` ($\leq$1.17.0)[7] directly copied the file of package `python-statistics`, which I assumed is a typical case of software ven-

[7]For a PyPI product, the version in the parentheses is the latest release in the corpus

doring. Also, product `dataspine-cli` (≤0.2.5) has the entire `python-boto3`. Yet, the copying is in a different manner. The package was copied to the subdirectory with path *dataspine-cli-0.2.4/venv/lib/python3.6/site-packages/boto3* along with other 70 other packages. For which, *venv* is a directory of the virtual environment that contains specific libraries that the Python interpreter require to run the program; the developer might included the virtual environment to let users install the dependencies themselves, or just by accident[8]. In the PyPI corpus, 4,620 releases in 1,867 products contain the directory *site-packages*, and the most frequent libraries inside are `pip` (in 72 products) and `setuptools` (in 71 products).

2. **Split or renamed product**: Package `python-jsonpath-rw` (1.4.0) and `pytest-tornado` (0.5.0) are identical cases, as their candidates `jsonpath-ng` (≤1.5.1) and `pytest-tornado5` (≤2.0.0) both have similar names and (almost) all of their identifiers ($incl(S, C)$ is 1 for the former and 0.98 for the latter). For `python-jsonpath-rw`, its package version is also the latest version of the target in PyPI. Same to the first reason, this package contains test files that are not in the target product `jsonpath-rw` (≤1.4.0). Noteworthily, this product was split into the candidate `jsonpath-ng` (≤1.5.1) that started from version `1.4.1`, and from then, it contains the test files in the package that was absent in the parent product. For another case, `pytest-tornado` has almost the same situation, so it is worth no more descriptions.

3. **Misleading ground truth**: The last 7 cases, starting from `python-uritemplate` (0.6), are those who had $incl(T, S)$ less than 0.1. For which, by manual inspections, I found no evidence that the targets are credible matches to the subjects. In this way, retrieving PyPI product names in Debian package setup files is not always a reliable method, although the best-effort choice. By coarse name searching on the PyPI website, I did not find possible equivalent products for these packages. However, package `kiwi` (1.9.22) is the only exception, which has nothing related to its unqualified target `kiwi` (≤9.21.7)[9] (probably a conflicting name), but it is identical to its candidate `kiwi-gtk` (≤3.0.3).

[8] In fact, the popular Python IDE PyCharm creates venv in the project root directory by default

[9] In the corpus, kiwi has 100 (the maximum number of downloaded) releases from 9.2.0 to 9.21.7, while kiwi started from 8.20.10 in the complete PyPI history (yet 1.9.22 is still not included)

6.4 Experiment 3: 3-filename Method

In this section, I define an alternative origin determination method, using at most 3 filenames of the Python files in each Debian package to match against filenames in the corpus. It is considered a more trivial approach because filenames are more superficial information of source code (not even in code) than identifiers. However, I expected that the filename method would produce comparable results to the 3-identifier fingerprint method because we proved that the product-level distinctiveness of filenames is even more than which of identifiers in the PyPI corpus. Thus, I would use the filename method as the baseline study, such that the proposed identifier method can be compared against and objectively evaluated.

6.4.1 Experiment Setting

The experiment setting of the filename method is analogous to which of the 3-identifier fingerprint method described in Section 6.2.1. In the sampling, filenames in a subject were filtered by a blocklist of the most frequent 32 filenames. I determined this size because the proportion of 32 filenames (over a total of 1.1 M filenames) equaled the ratio of 300 identifiers (size of the identifier blocklist used, over a total of 11.2 M identifiers) to make the comparison fair. Similar to the 3-identifier fingerprint method's sampling phase, if the number of filenames in a subject was less than 3 after the blocklist, the blocklist was no longer used.

I computed descriptive statistics about filenames in the subjects. Firstly, 351 subjects have only 1 or 2 filenames (in which one must be `setup.py`); by contrast, recall that only 9 subjects have 1 or 2 identifiers. The mean/median number of filenames per subject is 34.3/10. Secondly, suppose the 32-blocklist of filenames is applied always. In that case, 566 subjects will have 0 to 2 filenames left (only 21 subjects will have 0 to 2 identifiers after the 300-blocklist of identifiers), and the mean/median number of filenames per subject will decline to 31.7/8. The differences in the number of samples among the two methods imply that the comparison will be biased. However, the problem will later be addressed by subsetting the subjects.

In the matching phase, I obtained the number of PyPI products with all the sampled filenames, and I ordered the candidates with SourceRank the same way. The classification of the matching quality is also the same as before.

6.4.2 Experiment results - for all the 2,174 subjects

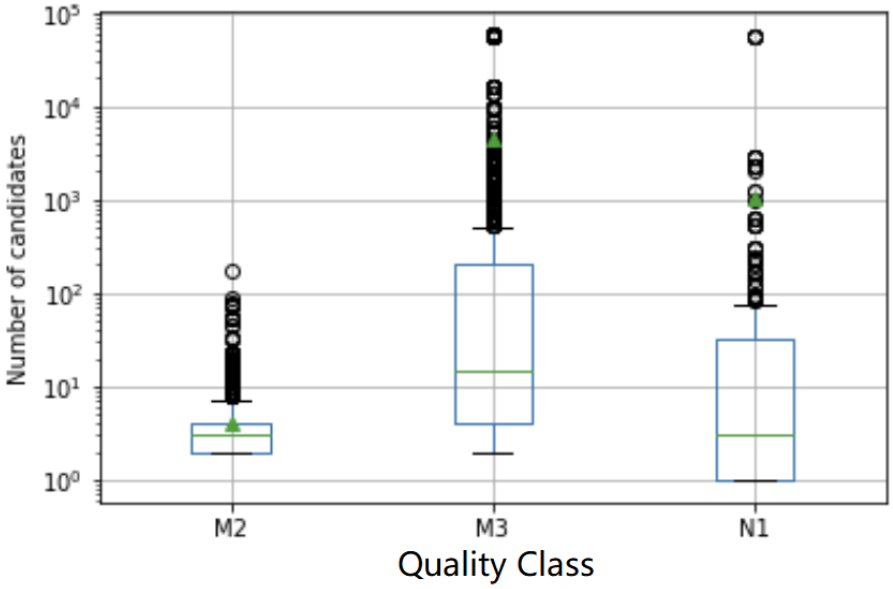
Table 6.3 documents the matching results using the matching quality classes (defined in Section 6.2.1) by both outcome and subject. Overall, the recall is 86.5%, which is a bit below the identifier method's (87.1%). However, the proportion of M1 outcomes is much lower (47.3% versus 54.6%). This method found 92.2% (100% - 1.6% - 6.2%) of the targets after all 5 iterations, which is similar to the identifier method (93.7%).

Table 6.3: The quality-classified results in both the outcome and subject dimension - the 3-filename method

Quality class	# Outcomes	# Subjects, any trial	# Subjects, 5 trials
M1 ($C = 1$, $R = 1$)	5,144 (47.3%)	1,250 (57.4%)	750 (34.5%)
M2 ($C > 1$, $R = 1$)	2,281 (21.0%)	686 (31.6%)	291 (13.4%)
M3 ($C > 1$, $R > 1$)	1,974 (18.2%)	570 (26.2%)	288 (13.2%)
N1 ($C > 0$, $R = 0$)	307 (2.8%)	108 (5.0%)	34 (1.6%)
N2 ($C = 0$)	1,164 (10.7%)	345 (15.9%)	135 (6.2%)

Also, in M2, M3, and N1, the mean/median rank of the target is 1310.0/4, contrasting to only 9.6/2 for the identifier method. Figure 6.5 shows the number of candidates in outcomes by the 3 quality classes. Particularly in M3, the mean/median number of candidates is as high as 4542.7/15, contrasting to only 59.3/5 for the identifier method.

Figure 6.5: The numbers of candidates for the outcomes in three quality classes - the 3-filename method



The filename method produced a comparable performance as the identifier method, only considering the recall. However, the numbers of candidates are very high (I suspect because many samples have less than 3 filenames). Therefore, I would like to focus on the results of both methods produced by *qualified samples*, which defines fingerprints or filename sets that have exactly 3 identifiers/filenames excluded from the corresponding blocklist, to re-compare the two methods.

6.4.3 Experiment results - for 1,608 subjects with qualified samples

I compared the two methods with a subset of the previous results. The subset has such property: each subject inside has at least 3 identifiers not in the 300-blocklist and at least 3 filenames not in the 32-blocklist. Such set of subjects has a size of 1,608.

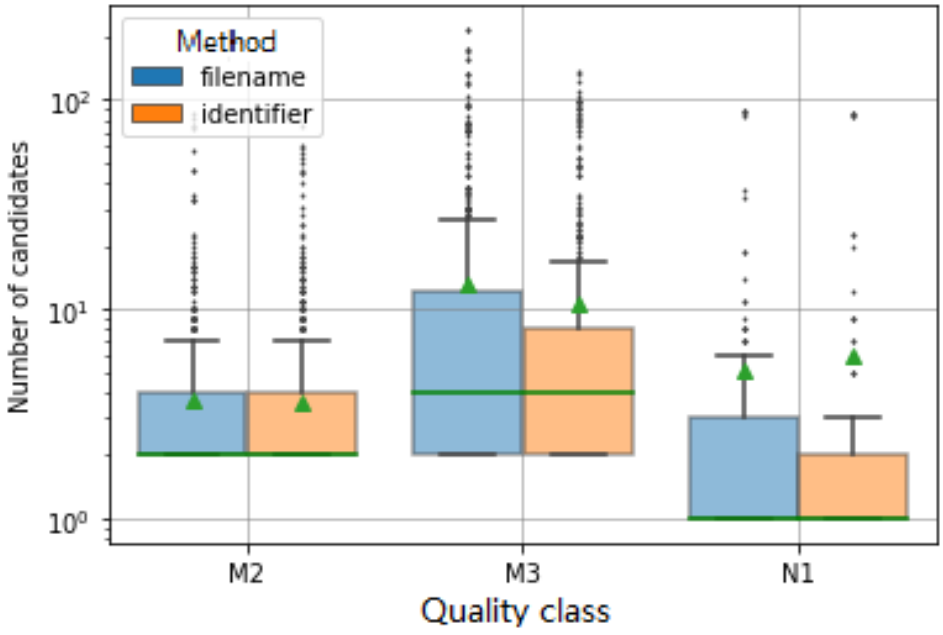
Table 6.4 documented the results by matching quality classes. In which the pair in each value field indicates corresponding results from the two methods, as (3-identifier/3-filenames). The results are similar to before. The recall is 86.3% for the identifier method and 85.2% for the filename method. All the recorded results are very close across the two methods.

Moreover, Figure 6.6 shows the number of candidates from each method. Recall that there were too many candidates when the unqualified samples (with less than 3 or frequent identifiers/filenames) were included, yet now the results determine that the filename method actually produces comparable numbers of candidates as the identifier method does. For M3, the mean/median rank is 2.9/2 for the identifier method and 3.3/2 for the filename method.

Table 6.4: The quality-classified results in both the outcome and subject dimension - the 3-identifier fingerprint method versus 3-filename method

Quality class	# Outcomes	# Subjects, any trial	# Subjects, 5 trials
M1 ($C = 1$, $R = 1$)	4,416/4,222 (54.9/52.5%)	1,132/1,056 (70.4/65.7%)	594/577 (36.9/35.9%)
M2 ($C > 1$, $R = 1$)	1,848/1,879 (23.4/23.4%)	599/590 (37.3/36.7%)	198/220 (12.3/13.7%)
M3 ($C > 1$, $R > 1$)	644/754 (8.0/9.4%)	288/309 (17.9/19.2%)	45/61 (2.8/3.8%)
N1 ($C > 0$, $R = 0$)	122/181 (1.5/2.3%)	58/76 (3.6/4.7%)	8/13 (0.5/0.8%)
N2 ($C = 0$)	981/1104 (12.2/12.5%)	321/309 (20.0/19.2%)	83/106 (5.2/6.6%)

Figure 6.6: The numbers of candidates for the outcomes in three quality classes - the 3-identifier fingerprint method versus 3-filename method



The two approaches have shared heuristics, using slightly different yet same-property information of software entities, so the results have a similar performance.

However, the filename method is still considered a more naive way of determining the origin because the filenames in software artifacts are way less diverse than identifiers, so it is not always practical (we have seen that many packages have less than 3 files). Besides, filenames are easier to lose during copying or when only partial contents are copied.

6.5 Summary

I have finished a series of empirical experiments to evaluate the proposed origin determination method by matching 2,174 selected "golden" Debian Python packages to products in the PyPI corpus. I used package names defined in setup files in the Debian packages as the ground truth (i.e., target product's name to compare the result with). First, I adopted the standard 3-identifier fingerprint method, which resulted in a recall of 87.1%, and the outcomes were exact matches 54.6% of the times. Hence, I answer the last research question by:

> **Answer to RQ 3:** The proposed method can effectively determine the product-level origin of Python source code entities sampled outside of the reference corpus. By experiments with only 3 non-frequent identifiers, the method finds the origin 87.1% of the times, in which 62.7% are exact matches. When the correct product is in more than one candidate, by SourceRank, it can be ordered to the top 67.8% of the time.

However, the method has a limitation, when the identifier used is not defined in the expected origin, the subject can not be matched. Due to this reason, 136 of the 2,174 packages were never matched in 5 trials of the method. Therefore, I designed a slightly more exhaustive method to improve the standard method, which uses all the identifiers in a subject to match candidates that include most of them. This method increased the recall to 96.6%. For the last 34 packages that were still failures, I conducted a qualitative study to identify 4 reasons: extra code in package, package copied by other products, split/renamed product, and misleading ground truth.

Finally, I performed a baseline experiment using filenames to match almost the same way as I did with identifiers. Since filenames have similar distinctiveness properties as identifiers do, the filename method performed just as the identifier method did. Regardless, filenames are not as many as identifiers.

Chapter 7

Discussion and Conclusion

This chapter provides a further discussion of the proposed method and experiment results, including the validity of the experiment methodologies, the limitations of the proposed method, and the future studies that are expected to benefit from my work. The last section concludes the thesis.

7.1 Threats to Validity

7.1.1 Construct validity

The construct validity mainly concerns building the reference corpus for the PyPI ecosystem. I regard downloading such an amount of packages and retrieving the identifiers as an engineering problem that consumes both computing power and human effort. I have conducted several integrity tests to ensure that the data was accurately processed at each stage. For example, I checked if the hash digests of the downloaded packages matched as they are documented in PyPI metadata. However, it was hard to verify if the decompression tools had ever failed to extract the complete contents of a package when I ran 100 jobs in parallel with Slurm. In fact, I have once countered a large-scale file system failure, as announced by the cloud platform Compute Canada Cedar cluster; during that, packages could not be decompressed and written to the file system, so I restarted the whole process since I could not identify how many packages were involved.

Also, I relied entirely on Universal Ctags for extracting class and function names in Python files, so the corpus's correctness depends on its reliability. Meanwhile, I compared the identifiers extracted with Python Abstract Syntax Tree with which from

using Ctags on several packages, and they turned out to produce identical results, so I trust Ctags. Similarly, I used SourceRank as the only metric to sort candidates, and I assumed that Libraries.io is well maintained, and the statistics for computing the metric were mainly correct (e.g., the number of releases, whether the metadata exists, the associated GitHub repository and its statistics, etc. for any product was accurate).

Another possible yet minor threat to construct validity is the decision of only downloading the latest 100 releases of every product in PyPI. Knowing that only 0.6% of the products had more than 100 releases and several products had too many large releases, I did it this way to economize the time and resource cost for building the corpus. I consider the impact of this decision is negligible because I expect recent releases have many identifiers that have appeared in old releases (even after 100 releases, since I have noticed some products, such as the mentioned one `CCXT` in Section , update frequently in configuration files rather than source code). However, in occasional cases when the subject entity is quite old (some were seen in the Debian golden set), we might fail to match the origin due to this decision.

7.1.2 Internal validity

Again, as part of the method, I used SourceRank to order the candidates equally likely to be the origin, which is a metric for the popularity and quality of products. Therefore, I naively assumed that if two products have a common entity, then this entity was copied (directly or immediately) from the more popular and quality one to another. However, I could also let the age of the candidates be a pivotal factor in ranking them, such that the candidate entity that has defined a fingerprint the earliest is more likely to be ranked first. Nevertheless, the main goal of the proposed method is to narrow down the search space rather than always return to the exact origin. Other research topics could address this problem further, such as analyzing the internal evolution of products or smaller entities from one to another in an OSS ecosystem.

Also, there exists some bias in the experiments. In the evaluation, as shown in Section 6.1, the SourceRank for products in the Debian subset is, on average higher than which for the entire PyPI corpus (since the Debian set is "golden"), which made it easier to put the correct origin at the top of the candidates. Moreover, in the experiment for determining the number of identifiers required to complete the method

effectively (as described in Section 5.3), only releases with 5 or more identifiers or files were sampled. However, having more identifiers might imply a product is less trivial, so it has created more original identifiers, which makes the sampled fingerprint more likely to be used to find the exact origin.

For another, I assumed PyPI is a trustworthy reference source for Python products. However, PyPI is pretty free to use, so developers cannot always ensure they upload the correct source code of their software. In some cases, I noticed developers included embedded dependencies in some of their packages released. For example, 1,867 products in the PyPI corpus had virtual environments, basically, as a directory named *site-packages* that contains source code of imported libraries in the. Without completely identifying and excluding identifiers in these embedded source code from the corpus, the proposed method might have performed with lower precision (e.g., returned more candidates).

7.1.3 External validity

The external validity mainly concerns with if the proposed method's properties are applicable outside my experiments' setting. On the one hand, the results only apply to source code entities written in Python and the PyPI ecosystem. I believe that identifiers in major programming languages and ecosystems are also distinctive across products; thus, I expect future studies that empirically analyze if similar identifier-based methods are universally applicable. It would also be interesting to investigate if identifiers are also generally distinctive across programming languages (some should be since we have observed that there are Python-specific naming conventions, such as identifiers starting with *_single_leading_underscore* as a weak "internal use" indicator, by definition in PEP 8).

On the other hand, I used the Debian golden set to evaluate the method, yet using packages from other distributions might produce different results. Debian packages are well maintained, so each should correctly contain the original source code. In this case, the situation might be different if the subjects are chosen from elsewhere.

7.2 Limitations

First, by our initial definitions, we could use the proposed method to match a subject entity to other entities in a reference corpus at various levels of granularity, such as

files. However, in this thesis, I only focus on the distinctiveness of identifiers at the product level, so I matched only products in the experiments. Generally, matching in finer granularities is not as effective as at the product level with the method. For example, identifiers must be less distinctive at the release level since different releases in a product mostly have many identifiers overlapped. As a possible solution, more information from source code, such as a wider variety of identifiers (e.g., including variables), might be helpful to uniquely identify files, which needs to be verified with extended empirical studies.

A significant drawback of the method's algorithm is that the origin must have all the identifiers in a sampled fingerprint to match. In the evaluation, we have witnessed many occurrences that sampled identifiers that were not in the original PyPI product, which resulted in failures. To deal with this case, I adopted a more exhaustive method that searches for entities with the most significant number of identifiers in the subject from the corpus. However, as the name suggests, such a method will no longer be lightweight when the subject has many identifiers. Therefore, a more adaptive algorithm combining both methods' advantages is in demand.

Lastly, I do not have sufficient evidence to claim that the candidate with the highest SourceRank is always the correct origin. I lack information on how the products are originated from one to another internally in an OSS ecosystem, which requires further research in this direction with similar or different methods.

7.3 Future Work

Several empirical studies and research topics could be pursued in the future. First of all, the same methodology could be used to analyze the distinctiveness of identifiers in other programming languages and ecosystems, then potentially, the generic method could perform similarly. The only requirement is creating a comprehensive reference corpus of identifiers for the desired language and ecosystem. The programming conventions and naming rules would potentially result in different levels of identifier uniqueness, which could impact the method's accuracy. It would be interesting to analyze the distinctiveness of identifiers when considering multiple corpora together. This way, we can determine a particular portion of identifiers that are pretty unique across languages, which helps origin determination when the language of the source code entity is unclear.

Secondly, a challenge that remains to be addressed is the method turning out low

accuracy when matching at a lower level of granularity, e.g., release, because releases from the same product tend to have a large portion of identifiers in common. As mentioned, it might be helpful to increase the types of identifiers (e.g., from **imported** modules/classes/functions, **called** methods, or defined variables), correspondingly the fingerprint size, that the method uses to match such types of entities. A large-scale empirical analysis is required to find the best choices and validate the effectiveness of using these types of identifiers.

Finally, the proposed method has a vast range of potential applications. A promising one is we could determine the origin of snippets rather than just one or more files. For example, Figure 7.1[1] shows a question posted on Stack Overflow Q&A website. In such context, the library name the questioner referred to was not mentioned in the title/text or by the dedicated classification tags. With the proposed method, identifiers can be automatically extracted from the code snippet and matched to products in our PyPI corpus. The identifiers *find_element_by_xpath*, *get*, *execute_script*, and filename *driver.py* (I assumed the imported driver module was inside the same library and not renamed as being imported) are only defined in 3 products `django-cloud-deploy`, `selene`, and `selene-kentastik`. Among these, `selene` (A Python binding for selenium WebDriver) is most likely to be the library the snippet referred to because of the highest SourceRank. Similar work has been done by Holmes et al. [72], who proposed a tool called Baker that integrates software resources of snippets, which is implemented as an add-on to Stack Overflow, as shown in Figure 7.2. Such a tool can be re-implemented for Python with the method and corpus I created. From another perspective, IDEs can benefit from the origin information to suggest (or automatically add) include/import statements.

[1]Titled: "Scraping tripadvisor review, len container change, no such element Unable to locate element," already removed by the questioner. `https://stackoverflow.com/questions/68878857`, lastly accessed 2022-01-16

Figure 7.1: Example of a Stack Overflow question that does not indicate which Python library it is using

I want to scrape reviews of tourist destinations on the tripadvisor web. I get this code. but when executed there is an error on certain pages, and len(container) which was 11, on the error page changes to 12. then, an error appears:

> no such element: Unable to locate element: {"method":"xpath","
> selector":".//div[@class='DrjyGw-P _26S7gyB4 _2nPM5Opx']"}

What should I add? please help me :(

Some code:

```
for i in range(from_page, num_page):
    print(f'halaman ke-{i+1}')

    # default tripadvisor website

    url = f"https://www.tripadvisor.co.id/Attraction_Review-g3177248-d2314079-Rev

    # if you pass the inputs in the command line
    if (len(sys.argv) == 4):
        path_to_file = sys.argv[1]
        num_page = int(sys.argv[2])
        url = sys.argv[3]

    # Import the webdriver
    driver.get(url)
    # expand the review
    time.sleep(5)
    element = driver.find_element_by_xpath("(//span[contains(@class, 'DrjyGw-P _1
    driver.execute_script("arguments[0].click();", element)

    first_container = driver.find_element_by_xpath(".//div[@class='_1c8_1ITO']")
    container = first_container.find_elements_by_xpath("./*")


    print(len(container))

    for j in range(len(container)-1):

        text_review = container[j].find_element_by_xpath(".//div[@class='DrjyGw-P
        review_text = text_review.find_element_by_xpath(".//span[@class='_2tsgCuq

        csvWriter.writerow((review_text,))
    print('Selanjutnya -->')
```

python web-scraping containers nosuchelementexception tripadvisor

Figure 7.2: Resource-augmented StackOverflow post, enabled by the tool Baker that Holmes et al. [72] developed to integrate software resources of snippets

7.4 Conclusion

Often, stakeholders in software development processes wish to know about the origin of their artifacts for the sake of resolving technical and ethical issues. In this thesis, I deal with a part of the origin determination problem: matching a subject entity (a file or a set of files) to a set of possible software product candidates. The proposed method only uses a few identifiers extracted from the subject, and the candidate set can be narrowed down to a handful size. Further, we can determine which candidate is most likely the correct product that the subject originated from with an empirical metric named SourceRank.

There are three main advantages of the method. First, since identifiers are concise information within source code, matching by identifiers is simple and fast. Second, the method is scalable to large OSS ecosystems, such as PyPI, and various levels of investigation granularity. Third, the method is robust to matching code with minor code modifications as long as the identifiers are preserved.

I have performed several empirical studies to prove the usefulness of the method. The results showed that, in the PyPI corpus, 75.8% of the class names and 76.7% of the function names are unique at the product level. After that, I determined a blocklist of the 300 most frequent identifiers that do not effectively help the matching. Then, I found that using only 3 identifiers excluding from this blocklist will match to the exact one product in nearly 80% of the time. Finally, the method was evaluated by determining the corresponding products of selected 2,174 Debian Python packages,

resulting in a recall of 87.1%; meanwhile, using SourceRank, the method successfully chose the correct origin among multiple candidates in 67.8% of the time.

Numerous research interests have been in related areas, such as code search and clone detection, which have slightly different purposes yet often involve similar techniques. The proposed method and the identifier corpus can potentially benefit various research topics. I expect a hybrid method, where the identifiers are used to narrow down the potential search space, and then more time-consuming algorithms such as clone detection are applied for finding the most confident match.

Bibliography

[1] Fact sheet: President signs executive order charting new course to improve the nation's cybersecurity and protect federal government networks. https://www.whitehouse.gov/briefing-room/statements-releases/2021/05/12/fact-sheet-president-signs-executive-order-charting-new-course-to-improve-the-nations-cybersecurity-and-protect-federal-government-networks/. Last accessed on 2022-10-13.

[2] Open source development and application security survey. https://securosis.com/assets/library/reports/Securosis_OpenSourceSurvey_Analysis.pdf. Last accessed on 2021-11-01.

[3] Pep 8 – style guide for python code. https://peps.python.org/pep-0008/. Last accessed on 2022-10-13.

[4] The python package index (pypi). https://pypi.org/. Last accessed on 2022-10-13.

[5] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 38–49, 2015.

[6] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377. IEEE, 1998.

[7] Khalid Belhajjame, Reza B'Far, James Cheney, Sam Coppens, Stephen Cresswell, Yolanda Gil, Paul Groth, Graham Klyne, Timothy Lebo, Jim McCusker, et al. Prov-dm: The prov data model. *W3C Recommendation*, 14:15–16, 2013.

[8] Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan I Maletic, Christopher Morrell, and Bonita Sharif. The impact of identifier style on effort and comprehension. *Empirical Software Engineering*, 18(2):219–276, 2013.

[9] RP Jagadeesh Chandra Bose, Kanchanjot Kaur Phokela, Vikrant Kaulgud, and Sanjay Podder. Blinker: A blockchain-enabled framework for software provenance. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pages 1–8. IEEE, 2019.

[10] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. Exploring the influence of identifier names on code quality: An empirical study. In *2010 14th European Conference on Software Maintenance and Reengineering*, pages 156–165. IEEE, 2010.

[11] Yang Cao, Christopher Jones, Vıctor Cuevas-Vicenttın, Matthew B Jones, Bertram Ludäscher, Timothy McPhillips, Paolo Missier, Christopher Schwalm, Peter Slaughter, Dave Vieglais, et al. Provone: extending prov to support the dataone scientific community. 2016.

[12] Bruno Caprile and Paolo Tonella. Restructuring program identifier names. In *icsm*, pages 97–107, 2000.

[13] Gabriella Castro Barbosa Costa Dalpra. *Supporting software processes analysis and decision-making using provenance data*. PhD thesis, Universidade Federal do Rio de Janeiro, 2018.

[14] Humberto LO Dalpra, Gabriella Castro Barbosa Costa, Tassio Ferenzini Martins Sirqueira, Regina MM Braga, Fernanda Campos, Cláudia Maria Lima Werner, and José Maria N David. Using ontology and data provenance to improve software processes. In *ONTOBRAS*, 2015.

[15] Ya Bin Dang, Ping Cheng, Lin Luo, and Adrian Cho. A code provenance management tool for ip-aware software development. In *Companion of the 30th international conference on Software engineering*, pages 975–976, 2008.

[16] Julius Davies, Daniel M German, Michael W Godfrey, and Abram Hindle. Software bertillonage. *Empirical Software Engineering*, 18(6):1195–1237, 2013.

[17] Florian Deissenboeck and Markus Pizka. Concise and consistent naming. *Software Quality Journal*, 14(3):261–282, 2006.

[18] Massimiliano Di Penta, Daniel M German, and Giuliano Antoniol. Identifying licensing of jar archives using a code-search approach. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 151–160. IEEE, 2010.

[19] Eric Enslen, Emily Hill, Lori Pollock, and K Vijay-Shanker. Mining source code to automatically split identifiers for software analysis. In *2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 71–80. IEEE, 2009.

[20] Qing Gao, Hansheng Zhang, Jie Wang, Yingfei Xiong, Lu Zhang, and Hong Mei. Fixing recurring crash bugs via analyzing q&a sites (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 307–318. IEEE, 2015.

[21] Mohammad Gharehyazie, Baishakhi Ray, and Vladimir Filkov. Some from here, some from there: Cross-project code reuse in github. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 291–301. IEEE, 2017.

[22] Michael W Godfrey. Understanding software artifact provenance. *Science of Computer Programming*, 97:86–90, 2015.

[23] Mark Grechanik, Chen Fu, Qing Xie, Collin McMillan, Denys Poshyvanyk, and Chad Cumby. A search engine for finding highly relevant applications. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 475–484, 2010.

[24] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 933–944. IEEE, 2018.

[25] Jacob A Harer, Louis Y Kim, Rebecca L Russell, Onur Ozdemir, Leonard R Kosta, Akshay Rangamani, Lei H Hamilton, Gabriel I Centeno, Jonathan R Key, Paul M Ellingwood, et al. Automated software vulnerability detection with machine learning. *arXiv preprint arXiv:1803.04497*, 2018.

[26] Johannes Hofmeister, Janet Siegmund, and Daniel V Holt. Shorter identifier names take longer to comprehend. In *2017 IEEE 24th International confer-*

ence on software analysis, evolution and reengineering (SANER), pages 217–227. IEEE, 2017.

[27] Reid Holmes and Gail C Murphy. Using structural context to recommend source code examples. In *Proceedings of the 27th international conference on Software engineering*, pages 117–125, 2005.

[28] Reid Holmes, Robert J Walker, and Gail C Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Transactions on Software Engineering*, 32(12):952–970, 2006.

[29] Einar W Høst and Bjarte M Østvold. Debugging method names. In *European Conference on Object-Oriented Programming*, pages 294–317. Springer, 2009.

[30] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. Binary code clone detection across architectures and compiling configurations. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 88–98. IEEE, 2017.

[31] Laura Inozemtseva, Siddharth Subramanian, and Reid Holmes. Integrating software project resources using source code identifiers. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 400–403, 2014.

[32] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*, pages 96–105. IEEE, 2007.

[33] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[34] Yalin Ke, Kathryn T Stolee, Claire Le Goues, and Yuriy Brun. Repairing programs with semantic code search (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 295–306. IEEE, 2015.

[35] Kennedy. Manage dependencies with godep. Technical report, Ardan labs, 2013. Last accessed on 2021-12-07.

[36] Kisub Kim, Dongsun Kim, Tegawendé F Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. Facoy: a code-to-code search engine. In *Proceedings of the 40th International Conference on Software Engineering*, pages 946–957, 2018.

[37] Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings Eighth Working Conference on Reverse Engineering*, pages 301–309. IEEE, 2001.

[38] Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Dire: A neural approach to decompiled identifier naming. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 628–639. IEEE, 2019.

[39] Dawn Lawrie, Henry Feild, and David Binkley. Syntactic identifier conciseness and consistency. In *2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 139–148. IEEE, 2006.

[40] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. What's in a name? a study of identifiers. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 3–12. IEEE, 2006.

[41] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. Effective identifier names for comprehension and memory. *Innovations in Systems and Software Engineering*, 3(4):303–318, 2007.

[42] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1):54–72, 2011.

[43] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on software Engineering*, 32(3):176–192, 2006.

[44] Erik Linstead, Sushil Bajracharya, Trung Ngo, Paul Rigor, Cristina Lopes, and Pierre Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery*, 18(2):300–336, 2009.

[45] Chao Liu, Chen Chen, Jiawei Han, and Philip S Yu. Gplag: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 872–881, 2006.

[46] Hui Liu, Qiurong Liu, Cristian-Alexandru Staicu, Michael Pradel, and Yue Luo. Nomen est omen: Exploring and exploiting similarities between argument and parameter names. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1063–1073, 2016.

[47] Kui Liu, Dongsun Kim, Tegawendé F Bissyandé, Taeyoung Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. Learning to spot and refactor inconsistent method names. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1–12. IEEE, 2019.

[48] Kui Liu, Dongsun Kim, Tegawendé F Bissyandé, Shin Yoo, and Yves Le Traon. Mining fix patterns for findbugs violations. *IEEE Transactions on Software Engineering*, 2018.

[49] Meili Lu, Xiaobing Sun, Shaowei Wang, David Lo, and Yucong Duan. Query expansion via wordnet for effective code search. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 545–549. IEEE, 2015.

[50] Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. Codehow: Effective code search based on api understanding and extended boolean model (e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 260–270. IEEE, 2015.

[51] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 111–120, 2011.

[52] Paolo Missier, Khalid Belhajjame, and James Cheney. The w3c prov family of specifications for modelling provenance metadata. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 773–776, 2013.

[53] Paolo Missier, Saumen Dey, Khalid Belhajjame, Víctor Cuevas-Vicenttín, and Bertram Ludäscher. D-prov: Extending the {PROV} provenance model with

workflow structure. In *5th {USENIX} Workshop on the Theory and Practice of Provenance (TaPP 13)*, 2013.

[54] Audris Mockus. Large-scale code reuse in open source software. In *First International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS'07: ICSE Workshops 2007)*, pages 7–7. IEEE, 2007.

[55] Luc Moreau, Ben Clifford, Juliana Freire, Joe Futrelle, Yolanda Gil, Paul Groth, Natalia Kwasnikowska, Simon Miles, Paolo Missier, Jim Myers, et al. The open provenance model core specification (v1. 1). *Future generation computer systems*, 27(6):743–756, 2011.

[56] Luc Moreau and Paul Groth. Provenance: an introduction to prov. *Synthesis lectures on the semantic web: theory and technology*, 3(4):1–129, 2013.

[57] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. How can i use this method? In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 880–890. IEEE, 2015.

[58] Son Nguyen, Hung Phan, Trinh Le, and Tien N Nguyen. Suggesting natural method names to check name consistencies. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1372–1384, 2020.

[59] Joel Ossher, Hitesh Sajnani, and Cristina Lopes. File cloning in open source java projects: The good, the bad, and the ugly. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 283–292. IEEE, 2011.

[60] Md Rizwan Parvez, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Building language models for text with named entities. *arXiv preprint arXiv:1805.04836*, 2018.

[61] Daniel Perez and Shigeru Chiba. Cross-language clone detection by learning over abstract syntax trees. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 518–528. IEEE, 2019.

[62] Anthony Peruma, Venera Arnaoudova, and Christian D Newman. Ideal: An open-source identifier name appraisal tool. *arXiv preprint arXiv:2107.08344*, 2021.

[63] João Felipe N Pimentel, Paolo Missier, Leonardo Murta, and Vanessa Braganholo. Versioned-prov: A prov extension to support mutable data entities. In *International Provenance and Annotation Workshop*, pages 87–100. Springer, 2018.

[64] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. Mining stackoverflow to turn the ide into a self-confident programming prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 102–111, 2014.

[65] Michael Pradel and Thomas R Gross. Detecting anomalies in the order of equally-typed method arguments. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 232–242, 2011.

[66] Michael Pradel and Koushik Sen. Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–25, 2018.

[67] Andrew Rice, Edward Aftandilian, Ciera Jaspan, Emily Johnston, Michael Pradel, and Yulissa Arroyo-Paredes. Detecting argument selection defects. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–22, 2017.

[68] Lawrence Rosen. *Open source licensing*, volume 692. Prentice Hall, 2005.

[69] Guillaume Rousseau, Roberto Di Cosmo, and Stefano Zacchiroli. Software provenance tracking at the scale of public source code. *Empirical Software Engineering*, 25:2930–2959, 2020.

[70] Andrea Schankin, Annika Berger, Daniel V Holt, Johannes C Hofmeister, Till Riedel, and Michael Beigl. Descriptive compound identifier names improve source code comprehension. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 31–3109. IEEE, 2018.

[71] Harry M Sneed. Object-oriented cobol recycling. In *Proceedings of WCRE'96: 4rd Working Conference on Reverse Engineering*, pages 169–178. IEEE, 1996.

[72] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. Live api documentation. In *Proceedings of the 36th International Conference on Software Engineering*, pages 643–652, 2014.

[73] Lianshan Sun, Jaehong Park, and Ravi Sandhu. Engineering access control policies for provenance-aware systems. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 285–292, 2013.

[74] Synopsys. 2020 open source security and risk analysis report (OSSRA). Technical report, Synopsys, 2020. Last accessed on 2021-11-01.

[75] Georg Von Krogh, Stefan Haefliger, Sebastian Spaeth, and Martin Wallin. Open source software: What we know (and do not know) about motivations to contribute. In *The DRUID Conference: 17 June 2008; Copenhagen, Denmark*, 2008.

[76] World Wide Web Consortium (W3C). Prov model primer. `https://www.w3.org/TR/2013/NOTE-prov-primer-20130430/`. Last accessed on 2021-10-20.

[77] World Wide Web Consortium (W3C). Prov-overview. `https://www.w3.org/TR/prov-overview/`. Last accessed on 2021-10-20.

[78] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 87–98. IEEE, 2016.

[79] Peng Xu and Arijit Sengupta. Provenance in software engineering-a configuration management view. *AMCIS 2005 Proceedings*, page 515, 2005.

[80] Yang Yuan and Yao Guo. Boreas: an accurate and scalable token-based approach to code clone detection. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 286–289, 2012.