The Journal of Systems and Software 211 (2024) 111973

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Deriving modernity signatures of codebases with static analysis ☆

Chris Admiraal [a], Wouter van den Brink [a], Marcus Gerhold [b], Vadim Zaytsev [a,b,*], Cristian Zubcu [a]

[a] Technical Computer Science, University of Twente, Enschede, The Netherlands
[b] Formal Methods & Tools, University of Twente, Enschede, The Netherlands

ARTICLE INFO

Keywords:
Coupled evolution
Software evolution
Programming language adoption

ABSTRACT

This paper addresses the problem of determining the modernity of software systems by analysing the use of new language features and their adoption over time. We propose the concept of *modernity signatures* to estimate the age of a codebase, naturally adjusted for maintenance practices, such that the modernity of a regularly updated system would be above that of a more recently created one which neglects current features and best practices. This can provide insights into coding practices, codebase health and the evolution of software languages. We present case studies on PHP and Python code, demonstrating the effectiveness of modernity signatures in determining the age of a codebase without executing the code or performing extensive human inspection. The paper describes the technical implementation details of generating the modernity signature for both of these languages, including the use of existing tools like the PHP parser and Vermin. The findings suggest that modernity signatures can aid developers in many ways from choosing whether to use a system or how to approach its maintenance, to assessing usefulness of a language feature, thus providing a valuable tool for source code analysis and manipulation.

1. Introduction

The concept of "modernity" has been a topic of discussion in many fields, including philosophy, sociology and technology. In the software engineering context, modernity can be defined as the extent to which the source code of a software system utilises new features and capabilities of the programming language it is written in. We will follow up in the next sections by formally defining what a modernity signature is before we run our experiments around it. Such experiments will involve analysing the source code of real existing systems in two software languages with a different history, to see how newly introduced features of those languages find their way into the code.

There are several reasons why investigating modernity is important. First of all, as programming languages evolve, they tend to introduce new features and capabilities with almost every release. The intention of language designers is to facilitate developers in writing more efficient and maintainable code, in a faster and less error-prone way. Therefore, analysing the use of new language features can provide insights into the coding practices of software developers and help identify areas where they may need further training or support. Making the opposite assumption of developers being ideally competent, we can also draw conclusions for the language designers, signalling back to them which features were worth implementing and which are, for one reason or another, still lacking community acceptance.

Secondly, the active use and adoption of new language features can also serve as an indicator of the overall health of a codebase. Intuitively, code that is actively maintained and updated, will also naturally use newer language features at least to some extent. And we already know since the time of Lehman (1980), that code that is regularly maintained, is also more likely to be robust, secure and scalable, while outdated and neglected code is more prone to bugs, security vulnerabilities and performance issues. In essence, lower modernity is indicative of the codebase moving towards becoming legacy.

Finally, analysing modernity in software systems can help us understand how technology is advancing and evolving over time. By identifying trends in the adoption of new features across not just codebases but also languages, we can gain insights into the direction of the software development discipline and the types of challenges that developers are facing.

To summarise, investigating modernity in software systems by analysing the active use of new language features is an important area of research that provides valuable insights into coding practices, codebase health, and the evolution of technology.

The contributions of the current document are as follows:

☆ Editor: Dr. Shane McIntosh.

* Corresponding author at: Formal Methods & Tools, University of Twente, Enschede, The Netherlands.
E-mail addresses: m.gerhold@utwente.nl (M. Gerhold), vadim@grammarware.net (V. Zaytsev).
URLs: https://mgerhold.personalweb.utwente.nl (M. Gerhold), http://grammarware.net (V. Zaytsev).

https://doi.org/10.1016/j.jss.2024.111973
Received 12 July 2023; Received in revised form 16 December 2023; Accepted 19 January 2024
Available online 3 February 2024

- We identify an important research direction and connect it to existing trends and activities in the domain of source code analysis and manipulation (Section 2).
- We propose a definition of modernity signatures, and a method for extracting them from versioned software repositories (Section 3).
- We demonstrate the effectiveness of the method through the use of two case studies, concerning PHP (Section 4) and Python (Section 5) code.
- We examine the effects that different normalisation techniques have on modernity signatures (Section 6).
- We summarise our findings with six lessons learnt (Section 7).

An earlier version of this work was already presented at the SCAM 2022 conference (van den Brink et al., 2022a), and Section 4 largely repeats the material from it. In Section 5, we describe another case study of comparable size, thoroughness and complexity, performed on Python code this time instead of PHP code. The accompanying artefact (Admiraal et al., 2023) is also written in a way to improve the replication experience as opposed to the original artefact (van den Brink et al., 2022b), in particular when concerning reproducibility and robustness (Benureau and Rougier, 2018). Namely, it includes a feature that allows the setting of a maximum release date. By choosing a date closer to the time of our study, it is possible to generate plots that match the original ones exactly. This outcome strongly affirms the reproducibility of our code and lays a solid groundwork for the subsequent stages of our investigation. In Section 6, we collect some thoughts and results on the choice of normalisation formulae used as the part of the algorithm. The paper is concluded by Section 7. The technical content of the paper is based on several student graduation projects (van den Brink, 2022; Admiraal, 2023a; Zubcu, 2023).

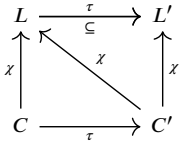
2. Related work

Estimating the age of a codebase is a known problem in software comprehension, useful for many purposes:

- Analysing `IDENTIFICATION DIVISION`s and generated comments is one of the first steps in industrial legacy codebase analysis, in order to determine the exact language (Kennedy van Dam and Zaytsev, 2016) and dialect (Lämmel and Verhoef, 2001) which define what tools are needed to handle the code.
- Age ranges are used to partition projects into new, enhancement and maintenance (Jones, 1995).
- For frontend systems, the age of code determines explorable vulnerabilities that it inevitably contains (Muscat, 2016).
- For comprehension purposes, the age influences coding idioms, programming style and required expertise (Farooq and Zaytsev, 2021b).
- For mergers and acquisitions, comparing modernities and styles of merging codebases leads to better cost estimates.
- The hazard function, failure rate, etc. in information management are age-specific and, depending on the age of the project, can be negligible or overwhelming (Verhoef, 2002).
- In quantitative IT portfolio management, the age of a project helps interpreting the data about its costs (accounting for inflation and currency reforms) (Verhoef, 2002).

In general, there are three important directions of related work worth pointing out before we zoom in on analysis of code in the chosen languages: (1) coevolution, (2) static modernity, and (3) dynamic modernity.

2.1. Coevolution



Technically, maintaining modernity relies on coupled evolution of code with the language it is written in Favre and Nguyen (2004), Lämmel (2016), Zaytsev (2016). The language L is transformed (τ) into the language L', and the code C which conformed (χ) to the original language L, now needs to be transformed (τ) into a modernised version of itself, C', which must conform (χ) to the updated language L'.

In practice the situation is somewhat more complex, since most languages evolve in a backwards-compatible way (more on this in Section 5), which means that the relation $C' \xrightarrow{\chi} L'$ is guaranteed by construction simpy because $C' \xrightarrow{\chi} L \subseteq L'$. Hence, we should be talking about conformance in a broader sense, taking into account languages' idioms, companies' coding conventions, developer teams' agreements, and many other aspects going beyond syntactic conformance. In a narrow sense, we can still isolate the new features of $L' \setminus L$ and speak of some form and degree of conformance to that part of the language. At the moment there exists no formal theory to support this (Lämmel, 2018).

2.2. Static modernity

Looking at the modernity signature of a version of a codebase presents a special useful case for tasks like language identification (Kennedy van Dam and Zaytsev, 2016) or version identification (Gerhold et al., 2023): given a codebase at a specific point in time, determine the language version for which it was written. This *static* modernity signature then represents a snapshot of a codebase. Since modernity concerns itself with a layer above simple syntactic conformance, we will quickly find investigating it being related to processes like smell detection (Sharma and Spinellis, 2018; Aljedaani et al., 2021). The current observable trend in smell detection research comprises attempts to let machine learning perform most of the work (Azeem et al., 2019; Fontana et al., 2013; Liu et al., 2021).

As one of the most recent projects, Zhang et al. (2022, 2023) have developed a tool that detects pythonicity of given code and proposes a refactoring in such a way that it was possible for the authors to submit a number of pull requests at various real world projects, and get them merged. Updated idiomatic code is known to be more concise (Ajami et al., 2019), secure (Heinl et al., 2020), performant (Sharma and Anwer, 2014), learnable (Wiese et al., 2017), lending itself easier to code review (Han et al., 2020) and migration (Mateos et al., 2019), etc. However, existing research adopts a code perspective and focuses on analysing known idioms instead of looking from the language perspective and analysing how much of the newly added features of a particular language version are used in the idioms found in the code.

2.3. Dynamic modernity

Since software systems continually evolve, their modernity changes too — and the applications we have hinted at above, indicate that simply assessing the modernity of one snapshot of a system might not be enough. By adding the spacial axis (literally, as we will demonstrate with our visualisations in Section 4 and Section 5), we can also compare the speed and the nature of change at the language level, with the speed and the nature of changes residing on the code level. In this case, we speak of *dynamic* modernity. There is some related work on smell evolution: for example, Kim (2020) recently analysed smells in test

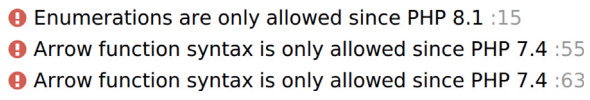
code and could formulate some concrete advice for testing engineers who want to improve their strategies. Even though code smells in test code have little effect on post-release defects (Kim et al., 2021), they are known to increase maintenance effort. Studying the evolution of idioms in the source code is also receiving increased attention: both Sakulniwat et al. (2019) and Farooq and Zaytsev (2021b) independently conclude that it is very common and natural for developers to refactor their code towards being more idiomatic, as time goes by.

As a concrete example, Farooq and Zaytsev (2021b) look into pythonicity (idiomaticness) of real world code. The more pythonic the code is, the more it conforms to the community standard and thus to the expectations of an average developer, minimising surprises and boosting maintainability. They discover that some projects have visually identifiable feature adoption patterns. The most prominent one was a pattern of a steady rise in use: the more time passes after a feature is introduced into the language, the more prominent the use of it becomes. Another pattern observed was that of an overly enthusiastic spiked adoption of a feature right after its release, promptly followed by a large scale abandoning of it (presumably because it ended up damaging the system design, understandability or performance). Finally, some dynamic modernity patterns looked like a "hype curve" with a slowly rising adoption, a peak of disappointment, a trough of disillusionment and an even slower second-generation adoption phase. In Farooq et al.'s work, as typical for this line of research, the code pythonicity is calculated per idiom, and in our work we specifically focus on each language feature one by one. One idiom may make use of several language features introduced at different points in time.

2.4. PHP

With PHP being a rather dynamic language, there is a spectrum of tools for it using dynamic analysis (Merlo et al., 2007; Papagiannis et al., 2011), static analysis or a combination thereof. Since we prefer to stay with static analysis as far as possible, the most related existing solutions for us are PhpStorm (JetBrains, 2022), PHP AiR (Hills and Klint, 2014; Hills et al., 2017) and the unnamed framework by Hauzar and Kofron (Hauzar and Kofron, 2015). Most PHP code analysis tools have a strong focus on standard enforcing (e.g., smell detection) and security (including instrumentation).

In the study by Hills et al. (2017), statistical analysis was performed on feature usage in multiple open source projects. Various interesting insights and conclusions come forward, but no comments are made on the modernity, or lack thereof, of the code in the corpus. Current efforts in analysing PHP systems to determine its age are mostly focused on determining the language level in terms of compatibility. For example, PhpStorm (JetBrains, 2022) contains a static analysis tool to determine whether language features are used which are not supported by the minimum version specified by the developer:



Another example is PHP Compatinfo (Laville, 2022), a tool which for any given system determines the minimum required PHP version and installed extensions. The tool is mature, and gives more information on the required PHP version than PhpStorm. However, it does not give information on the age of the codebase it analyses as a whole.

In general, statistical analysis of languages and programs is a much more popular topic in natural language processing (mostly with tree adjoining grammars), but there exist attempts to use such techniques for language translation (Karaivanov et al., 2014). We are not aware of any projects using construct usage based modernity signatures to analyse and/or compare software systems.

2.5. Python

Modernity of a Python project is often related to how *pythonic* its code is — with the notion of "pythonicity" being deeply engraved in the community and changing over time (Farooq and Zaytsev, 2021b; Alexandru et al., 2018). The relation between pythonic idioms as "reusable abstractions" in the code, and the everyday practice of developing in Python, has first been studied by Alexandru et al. in 2018 (Alexandru et al., 2018). Structured interviews they conducted, were supported by literature research by Farooq and Zaytsev (2021b). Both converged on the same point: Python developers are fully aware of the evasive concept of pythonicity, and they apply it in a conscious and targeted way. Many of such idioms are also language features, introduced in a specific Python version, while some others are library functions or preferences among language features. To give a concrete example, formatting strings with `%` was seen as pythonic in the early years of the language's existence, but was slowly phased out by `str.format` and later with `f`-strings (Farooq and Zaytsev, 2021a).

The idiom detector by Farooq and Zaytsev makes use of the built-in `ast` module (Farooq and Zaytsev, 2021b). This is a popular way, but there exist many alternatives that use custom frontends, such as those based on ANTLR (Vavrová and Zaytsev, 2017) or Rascal (Vinju, 2021). Other related questions that are being asked about Python codebases are what is the impact of the application domain on the choice of language features (Peng et al., 2021) or which code smells are prominent in Python code (Chen et al., 2016) and how different are the smells found in Python code compared to code in other languages (Vavrová and Zaytsev, 2017).

2.6. Normalisation

The derivation of modernity signatures includes a *normalisation* step. For instance, this could be the weighing of nodes in an abstract syntax tree, or the individual size of a file in a software project, measured either absolutely or relatively. Naturally, this choice impacts the calculated modernity signatures. When addressing the topic of normalisation, it is noteworthy that existing research has compared various techniques across a range of domains. For instance, Chakraborty et al. evaluated the effect of normalisation in the context of Multi-Attribute Decision Making (MADM) problems (Chakraborty and Yeh, 2009). In another study, Dubois et al. highlighted the significance of normalisation in scaling protein data for medical research (Dubois et al., 2020). These studies demonstrate the pervasive importance of normalisation techniques in different research contexts. However, due to the novelty of the topic, a research gap exists when it comes to the application of normalisation specifically to modernity signatures.

3. Modernity signatures

We define **modernity** as a scale of measuring the age of a codebase. As a proxy for that age, we come to the abstraction of a **language level**, which is a minor release of the language and its default official compiler (e.g., PHP 8.0 or Python 3.11 or 5697-V61 IBM Enterprise COBOL Version 6 Release 4), since it determines which minimal version of the compiler must be installed to run the code. In the context of a legacy codebase of COBOL generated from a 4GL (Zaytsev and Fabry, 2019), it could mean phases of the 4GL's existence. For example, for a language currently known as CA Gen, "IEF" in the first comment dates the code back to 1990–1996, "Composer" to 1996–1997, "COOL:GEN" to 1997–2004, "Advantage Gen" to 2004–2012 and "CA Gen" to 2012+, while the language largely stayed the same (C.A. Technologies, 2016).

3.1. Motivating example

In general, our **modernity signature** takes the form of an n-tuple with n being the number of language levels we consider. A codebase has a modernity equal to a language level N if and only if the official compiler for version N is sufficient to compile the entire codebase, but the compiler for version $N-1$ is insufficient. For our case studies it will be the 13-tuple for PHP (Section 4) and 20-tuple for Python (Section 5). Every element of this tuple is supposed to be a quantitative representation of the extent to which the analysed code base looks like code written to be executed by this language level.

The modernity signature will use grammar usage statistics to derive the modernity of a code base. Consider, for example, the following grammar definition, which defines the grammar for attributed statements in the PHP language.

```
attributed_statement ::=
        function_declaration_statement
|       class_declaration_statement
|       trait_declaration_statement
|       interface_declaration_statement
|       enum_declaration_statement ;
```

The grammar defines an attributed statement to be the declaration of either a function, a class, a trait, an interface or an enumeration. In other words, there are five possibilities to choose from when creating an attributed statement in this grammar. By analysing a PHP system, we can add annotations to the grammar of how often the different paths are taken:

```
attributed_statement ::=
  [33%] function_declaration_statement
| [48%] class_declaration_statement
|  [1%] trait_declaration_statement
| [11%] interface_declaration_statement
|  [7%] enum_declaration_statement ;
```

In this example, enumerations are chosen in 7% of the cases, but enumerations were only introduced in PHP 8.1.0. Thus, we know that this code base will require at least language level 8.1. By adding metadata on the language level associated with the different paths in the grammar, and by adding usage statistics to the different possibilities offered by the grammar, we can infer the modernity signature of a PHP code base.

3.2. Formal definition

A modernity signature provides a quantitative measure of the extent to which a codebase utilises features from different versions of its programming language. Formally, we define the modernity signature of a codebase as an N-tuple,

$$M = (m_1, m_2, \dots, m_N),$$

where each element m_i corresponds to a version V_i of the language. It thus represents a quantifiable degree of usage of features specific to that version within the codebase.

To construct a modernity signature, we first catalogue the features introduced in each version of the language. We then parse the codebase to identify instances of these features. Each feature F is associated with the minimum version V_{min} that supports it. Hence, the presence of feature F in the codebase contributes to the calculation of the modernity signature by incrementing the value of m_{min}, reflecting that the codebase can only run on version V_{min} or higher.

The contribution of each feature to the modernity signature is then normalised over its usage within the codebase. We discuss the effect that different normalisation techniques have on modernity signatures in Section 6. For instance, the contribution of each feature can be weighted by the relative frequency or percentage of that feature's usage within the codebase. If a feature is used frequently, it indicates a stronger reliance on the language version that introduced the feature. Hence, the modernity signature not only captures the presence of language features but also their prevalence within the codebase.

Lastly, the overall modernity signature M is the aggregation of the contributions for all features F in the codebase. The overall workflow is described in Fig. 1.

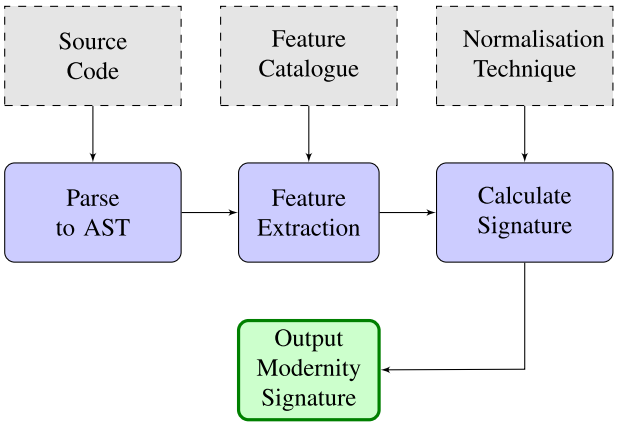


Fig. 1. Schematic overview describing the workflow to calculate the modernity signature of a codebase. First the input source code is parsed to its abstract syntax tree. Next features are extracted based on a catalogue of features, e.g., the Python feature documentation or a PHP annotated grammar. Then the modernity signature can be calculated and normalised based on a normalisation technique.

4. Case study I: Modernity signatures of PHP systems

In its long history and many versions, the PHP language has undergone many changes. One of the first versions of PHP used a Perl-like syntax in HTML comments (The P.H.P. Documentation Group, 2022). The rewrite of the language by Andi Gutmans and Zeev Suraski into an extensible language made it possible for other developers to add new functionality to the language, either by modifying its syntax or by adding new functions and data types. The language is still evolving nowadays, with the most recent development being the release of PHP 8.2 in November 2022 at the time of writing. This version adds many major additions to the syntax such as enumerations (Garfield and Tovilo, 2020) and intersection types (Banyard, 2021). These syntax modifications encourage PHP programmers to use new programming paradigms in their code. Other adjustments introduced by new language versions do not change the syntax, but rather modify the available functions and their signatures. For example, PHP 8.0 introduced the `str_contains()`, `str_starts_with()` and `str_ends_with()` functions (The PHP group, 2020). There exists a continuing migration from resource types to standard class objects, further elaborated by Karunaratne (2020).

4.1. PHP language levels

For every PHP system, we can define its language level as the minimum major PHP version required to be able to run the code in the system. For example, version 9.11.0 of the Laravel framework (Otwell, 2011) requires PHP version 8.0.2 or higher. The language level is then PHP 8.0. Today, information about the minimum required PHP version and other requirements imposed by a PHP system is usually contained in a `composer.json` file, an artefact produced by the Composer package manager, available at https://getcomposer.org.

The PHP language level indicated in the `composer.json` file by means of the minimum required PHP version does what it says on the

```php
public function exampleMethod() {
    // A Name node in the AST
    $name = "example";

    // An Identifier in early PHP versions
    echo $name;

    // An Expression in later PHP versions
    $expression = $name . " concatenated";
}
```

Fig. 2. Example PHP snippet for modernity signature calculation. The variable `$name` is used as an identifier at first, as an identifier and expression next and as an expression last, depending on the version of the language.

tin: it tells other developers wishing to use a system what version of PHP they should install to run the code. However, it does not tell much about the *actual* modernity, or rather, the age, of the codebase. While PHP regularly has backwards incompatible changes between major versions, much legacy PHP code will still run without problems in later PHP versions, or will do so with a few minor modifications.

As a result, it is possible to advertise a codebase as being compatible with a recent version of PHP, thereby implying that the system has been recently maintained, while most of the code is, in fact, very old and might contain several bugs and security issues. The actual modernity of the code is thus invisible to users of the system without performing extensive analysis. Thus, we wish to reliably determine the modernity of a PHP codebase without needing to execute the code, and without extensive human inspection.

Our implementation supports 13 major versions of PHP: **5.2** (the oldest relevant version before features intended for 6.x started to be introduced), **5.3**, **5.4**, **5.5**, **5.6**, **7.0** (there was no 6.0 after all, since the community decided that the abundant preceding discussions gave that version a bad reputation before it had a chance to be released), **7.1**, **7.2**, **7.3**, **7.4**, **8.0** (the oldest version currently in security support phase), **8.1** (the oldest version currently in active support phase) and **8.2** (the newest version at the moment of performing the case study; the version current at the moment of paper submission is one minor step further: 8.3[1]).

4.2. Technical implementation details

To calculate the modernity signature we built a tool on top of an existing PHP parser. We have used the PHP Parser from https://github.com/nikic/PHP-Parser which is the most up to date existing parser of PHP at the time of writing. It is thus capable of dealing with all possible versions of PHP that we have studied, to build our own library on. We parse the code and annotate each node within the generated parse trees with a range – defined by minimum and maximum values – for performance optimisation, representing the span of language levels that accommodate that particular construct. Determining these ranges was straightforward given that we know which language version introduced which constructs to the language. To optimise signature clarity, we employed many creative steps that even involved considering the exclusion of features introduced in the earliest language version supported, PHP 5.2. This design and tweaking happened only while applying our tool to the projects from the training part of the corpus. The corpus is discussed below in Section 4.3.

Thus, each leaf of every AST gets a range of language versions assigned to it. Once this data is accumulated per AST node type, the process gets more involved. We illustrate it by the following simplified example on Fig. 2. Suppose that a particular node `Name` in a tree (1) could only be an `Identifier` in one version of a language, (2) could be either `Identifier` or an `Expression` in the second version, and (3) only an `Expression` in the latest version of the language. Focusing on these three levels, our tool from six occurrences of a `Name` could have collected tuples $\langle 4, 4, 0\rangle$ for the `Identifier` which occurs four times across the tree, and $\langle 0, 2, 2\rangle$ for the `Expression`. We normalise both tuples such that their maximum values become 1 and lower ones are scaled down proportionally. We calculate the resulting signature for `Name` as follows:

$$^4/_6 \cdot \langle 1, 1, 0\rangle + {}^2/_6 \cdot \langle 0, 1, 1\rangle = \langle {}^2/_3, 1, {}^1/_3\rangle$$

If a `Name` occurs in another context next to a node `Type` with normalised signature $\langle {}^1/_2, {}^1/_3, 1\rangle$, and there are 30 `Name` nodes and 20 of `Type` nodes within some parent context, then the signature of that context will be subsequently calculated as:

$$^{30}/_{50} \cdot \langle {}^2/_3, 1, {}^1/_3\rangle + {}^{20}/_{50} \cdot \langle {}^1/_2, {}^1/_3, 1\rangle = \langle {}^3/_5, {}^{11}/_{15}, {}^3/_5\rangle$$

This signature gets normalised as $\langle {}^9/_{11}, 1, {}^9/_{11}\rangle$, and the calculation continues. We continue traversing the nonterminals, i.e. AST node types, and keep all counters normalised such that the maximum value in each is 1. We then multiply collected language level estimations of lower nodes by how likely such nodes were to occur according to the observed code. Eventually, the weighted sum is calculated for the entire AST. The weight of the tuples is still equal to the distribution of the occurrence of the node type. In this weighted sum, the tuples are normalised before the weighted sum is calculated. The effect of normalisation is discussed in detail in Section 6. To calculate the signature for an entire PHP system, the weighted sum of the normalised tuples is calculated in a similar way. Thus, by taking the weighted sum step-by-step we calculate the modernity signature of one PHP file. To calculate the modernity of an entire *project* containing multiple PHP files instead, we relate the signature of individual files to others by resorting to a file's size in bytes. For example, the tuple of a file of 500 bytes in a directory with 5000 bytes of PHP code will be granted a weight of 0.1 in the sum.

Our implementation is open source and can be explored at https://github.com/grammarware/modernity-php, respecting the open source MIT license. We welcome forks and accept pull requests.

4.3. Corpus construction

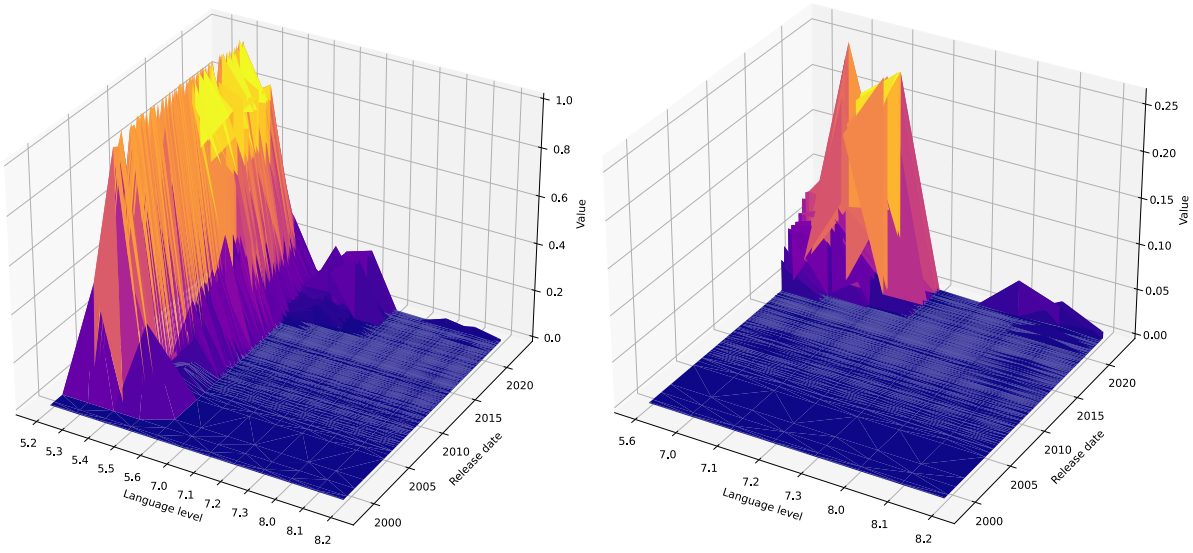
There is no available curated corpus of PHP systems readily available for mining/evolution research. Thus, we assembled a new corpus with two parts: the training set and the testing set. It consists of well-known PHP projects of which many historical versions were available. The corpus can be found in Table 1. All versions of all systems (299 in total) were annotated with the time of their release (either available explicitly or from the first `git` commit in the release tag) as belonging to a particular PHP level.

The repositories in the top of Table 1 were used for developing the modernity signature that could estimate the age of the system, and visualisations obtained from them were actively used to develop and tweak the definition of the signature based on matching calculated signatures to system release dates; and the ones in the table below were used to verify how well those signatures work. We did not change the design of the modernity signature after the testing set was used.

4.4. Results and discussion

The training phase showed us that the signatures tend to be heavily lopsided, independent of the release date of the software — accumulated results can be seen in Fig. 3(a). This is probably due to the fact that most node types in the PHP language have been supported since the earliest versions, no matter their form. To accentuate this,

[1] PHP 8.3.0 Released!, https://www.php.net/archive/2023.php#2023-11-23-2.



(a) With earliest levels **(b)** Without earliest levels

Fig. 3. Modernity signatures in the training phase, with and without signature values for the earliest language levels.

Table 1
Corpus of the PHP case study. We hand-picked popular PHP projects that had many historical versions available. Of those, we only considered stable minor version releases. The table at the top contains the training set that was used to optimise our technical implementation, while the bottom one shows the test projects we tested it on.

(a) PHP corpus training projects.

Project	Home page	Versions
CakePHP	https://cakephp.org	29
Joomla!	https://www.joomla.org	18
Laravel	https://laravel.com	16
phpMyAdmin	https://www.phpmyadmin.net	22
Symfony	https://symfony.com	32
WordPress	https://wordpress.org	44

(b) PHP corpus test projects.

Project	Home page	Versions
CodeIgniter	https://codeigniter.com	8
Guzzle HTTP	https://docs.guzzlephp.org	37
Monolog	https://seldaek.github.io	36
PHPUnit	https://phpunit.de	57

we present the signatures without the values representing these older language levels in Fig. 3(b). The signatures calculated in the training phase grouped by project are shown in Fig. 4 and those of the testing phase can be seen in Fig. 5. Here, too, the values corresponding to the earliest language levels have been omitted.

Figs. 4 and 5 highlight different patterns of signature change over time. An unbiased signature would have a similar shape for two different systems developed in the same year, but unfortunately, this is not the case. We conclude that there exists a bias in our signature. This can be partly explained by the nature of the projects in the corpus. We remark that different project authors have different strategies for maintaining their products. Compare, for example, the charts for Joomla! and Laravel in Fig. 4(b) and Fig. 4(c), respectively. It becomes apparent that the Joomla! team seems to strongly prefer supporting as much PHP versions as possible, while newer Laravel versions use newer PHP features.

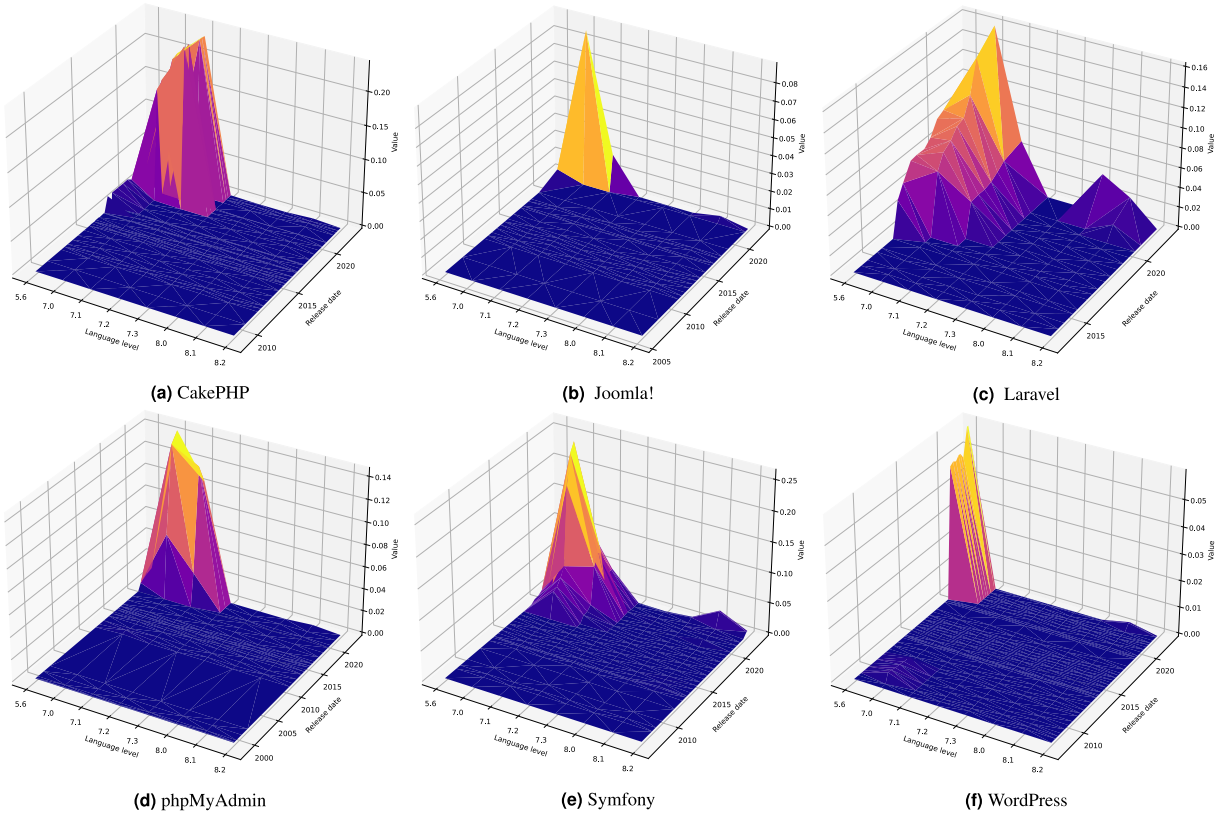
This change in adoption of newer language levels can be explained by the intended user base of the software. Typically, users of Joomla! use it on a shared web hosting environment where they do not control the installed PHP version and need to deal with dependencies on a plethora of extensions (some of which are legacy on their own). Unfortunately, older versions are still actively in use today (Roose, 2022), and Joomla! developers have to account for this. On the other hand, Laravel is mostly used by "web artisans" (the term used on the main website of Laravel) with full access to the server on which their project is deployed. One could argue that Laravel developers will thus prefer to use the latest stable PHP version and the features it introduces.

For most projects in the corpus, there is a strong connection between the release date of a PHP system and the shape of the modernity signature. The key exception is Joomla!, where this pattern emerges from the versions released after 2021. The exact nature of the connection differs significantly among projects. Laravel, for example, starts showing skewed signatures as early as 2015, while WordPress only starts behaving like this in 2020. We conclude that the signature is able to determine the relative age of a PHP system, but is not indicative of an absolute release date. It remains to be investigated how much of this observation is based (a) on the very concept of modernity based on grammar usage statistics, (b) on the peculiarities of PHP, (c) on noise and biases in our corpus.

4.5. Conclusions for PHP

We built a prototype for calculating modernity signatures for PHP code, available at https://github.com/grammarware/modernity-php. Our tool uses an existing PHP parser, implements several AST visitors, and uses weighted sums to converge to one n-tuple of version support estimations where n is the number of PHP versions from 5.2 to 8.2 (13 versions total).

The results of Section 4.4 show different modernity signature patterns emerging. One such pattern aims at providing a whole range of features, while another seems quick to adapt new language features. To illustrate, Joomla! supports a wide variety of extensions and dependencies, cf. Fig. 4(b). Generally, Joomla! users use it on shared web hosting environments where they do not controlled the installed PHP versions. It is thus paramount that Joomla! supports many features. Conversely, Laravel generally supports newer PHP features, cf. Fig. 4(c). We conjecture this is due to Laravel's intended user bases that primarily focuses on "web artisans" with full access to the server their project is based on. The curves of the other modernity signatures in the training and testing phase seem to largely follow a similar structure, cf. Figs. 4 and 5. This very consideration highlights one of the benefits of modernity signatures: They allow inference on a project's intended field of usage. While a deeper comparison would undoubtedly provide additional clarity, it goes beyond the intended scope here.



(a) CakePHP **(b)** Joomla! **(c)** Laravel

(d) phpMyAdmin **(e)** Symfony **(f)** WordPress

Fig. 4. Modernity signatures calculated in the training phase, grouped by project, without signature values for the earliest language levels.

5. Case study II: Modernity signatures of Python systems

Python is a programming language initially developed by Guido van Rossum (van Rossum and Drake Jr., 1995) and first released in 1991. By 2023 it has reached the top position in the TIOBE index of programming language popularity (TIOBE, 2023). It is known for its simplicity, versatility and readability. Since its initial inception it has evolved from a niche scripting language to a fully-fledged programming language used in a wide variety of domains; Python is used in data science (Idris, 2014; McKinney, 2012; Nelli, 2015), artificial intelligence (Raschka and Mirjalili, 2019) and web development (Forcier et al., 2008; Grinberg, 2018; Challapalli et al., 2021). Due to its ease of use it is the entry-level language of many new programmers, and its continued popularity can be partially attributed to its versatile arsenal of off-the-shelf libraries.

Over the years, Python has continued to evolve and mature with frequent updates and new versions, both major and minor. One of the language's key strength is its ability to maintain backwards compatibility in minor version updates while simultaneously being able to introduce new features. For the most part, this has allowed developers to write modern and maintainable code using Python without having to worry about deprecated features or breaking changes.

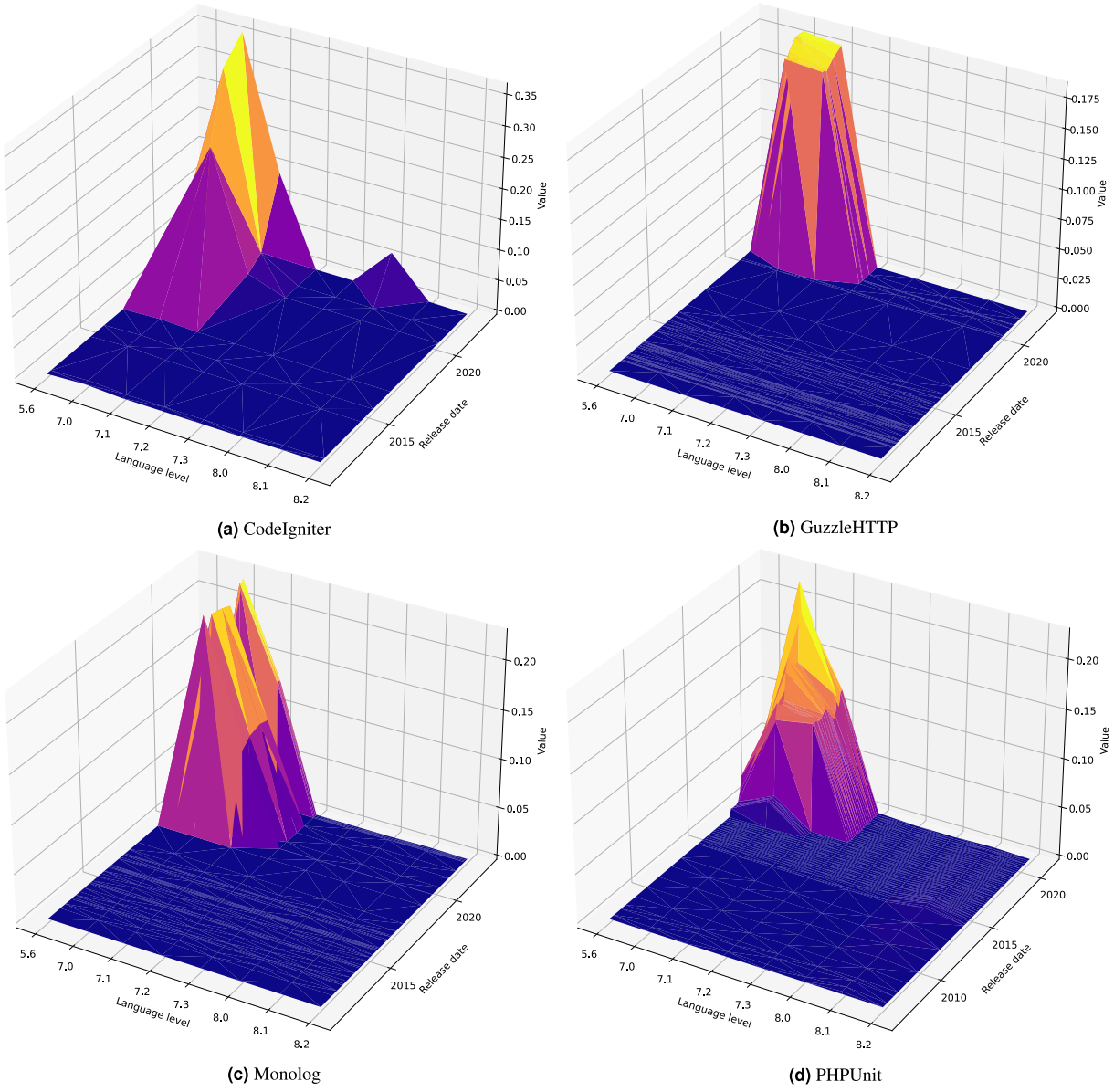
5.1. Python language levels

Python underwent many changes throughout the past 20 years. Undoubtedly, one of the most breaking changes occurred when major version 3 released (Python, 2008). Python 3 introduced many key improvements, but also critically broke backwards compatibility in some cases. For example: (1) `print` was not a keyword anymore, but a function; (2) the Boolean values *True* and *False* are now reserved keywords; and (3) strings were Unicode by default instead of ASCII. In the release of Python 3.10 new syntax features were added, the most notable being Structural Pattern Matching (Kohn and van Rossum, 2020) which works similar to `switch-case` statements in other programming languages. All these new syntax features allow programmers to write their code structurally differently.

However, having many backwards compatible syntax changes to the language implies that there are multiple ways of writing code that functionally behaves the same. In this section we focus on this difference to give an indication about how old, and perhaps therefore arguably also how maintained individual Python projects are.

In Section 4 we constructed one grammar usable for all PHP versions and used it to generate an Abstract Syntax Tree (AST) to generate the modernity signature of PHP projects. This approach does not carry over directly to Python. A problem arises when we want to compare grammars of both Python 2 and Python 3 code. Only considering either version is undesirable, since we can see how many features that were introduced in Python 2, are still used nowadays. To compare both versions, we would like to have one grammar that parses both languages. Grammars that parse Python 2 and Python 3.0 till 3.9 do already exist. For example, Vavrová and Zaytsev used a combination of two existing ANTLR grammars, but the Python 3 grammar has not been updated at this point in time (Vavrová and Zaytsev, 2017; dours, 2021). Also, third party grammars such as *Parso*, which parse both Python 2 and 3 code into the same AST, have not succeeded to do so (Taskaya, 2020). This AST differs from the built-in `ast` Python module (Python, 2022).

The main reason these grammars do not support Python versions 3.9 and above, is the introduction of the Parsing Expression Grammar (PEG), which replaced the LL(1)-based parser. A PEG grammar is distinct from a context-free grammar, such as the old Python grammar, because it more closely reflects the parser's behaviour during parsing. The crucial difference is moving from a symmetric choice operator (also known as "the BNF bar") to an ordered choice among production rules. With the introduction of the new keywords `match` and `case` in Python 3.10 (Python, 2021), they were turned into *soft keywords*. This means that they are recognised as keywords at the beginning



(a) CodeIgniter **(b)** GuzzleHTTP

(c) Monolog **(d)** PHPUnit

Fig. 5. Modernity signatures calculated in the testing phase, grouped by project, without signature values for the earliest language levels.

of a match statement or case block but are allowed to be used in other contexts as variable or argument names (Bucher et al., 2020). As a result, the current Python grammar may not be easily written as an LL(1) grammar. Presumably this PEG-problem could be solved by convergence of both major grammars (Zaytsev, 2011; Lämmel and Zaytsev, 2009), which has been done before with Python (Malloy and Power, 2018). As this is not the focus here, we sidestep the (admittedly interesting and challenging) issue of modelling soft keyword support in non-PEG grammars by relying on Vermin recovery and fallback mechanisms.

5.2. Technical implementation details

With modernity as a scale of measuring the age of a project, the modernity signature should take the form of an n-tuple with n being the number of minor Python 2 & 3 versions. At the time of writing, there are 20 such versions available: **2.0–2.7** and **3.0–3.11**. Every version is thus an element in a tuple with 20 entries, cf. Table 2. Conversely, every element in this tuple is a number that represents how many features introduced in this Python version are used in a project. To compare different releases of the same project, we normalise this tuple by dividing all its elements by the total sum of this tuple. The effect of normalisation is discussed in detail in Section 6.

To generate the modernity signature of any given project, we need to know how many features per Python version it uses. Fortuitously, a tool that achieves this exists: `Vermin` Kristensen (2018) — a portmanteau of "Version" and "Minimum". `Vermin` claims to be able detect the minimum Python version, major and minor, needed to run any given Python code. It is also able to tell us why it requires a certain version by returning a list of features. It achieves this by traversing the built AST and detecting if predefined rules for features match.

To know how many features per Python version are in a project, we proceed as follows: (1) Download the project from PyPI and only extract all Python files. Projects that do not contain Python files are not considered in our corpus. For instance, this can be the case if a project is mainly written in another programming language like C, or when only its build distribution is available and not its source distribution. (2) Iterate over all these files and call `Vermin` to return all features with the Python version they were introduced in these files. Notably, some features are introduced in two different Python versions,

Table 2
Verification failures per version. The top table shows how many test cases we generated based on the documentation per language version. The bottom table shows how many times, in absolute numbers and fraction-wise, does Vermin report a different version than the one we expect.

Python version	**2.0**	**2.1**	**2.2**	**2.3**	**2.4**	**2.5**	**2.6**	**2.7**
Tests generated per version	99	199	120	297	223	270	444	279

3.0	**3.1**	**3.2**	**3.3**	**3.4**	**3.5**	**3.6**	**3.7**	**3.8**	**3.9**	**3.10**	**3.11**
32	67	282	313	285	195	98	185	178	88	113	110

Python version	**2.0**	**2.1**	**2.2**	**2.3**	**2.4**	**2.5**	**2.6**	**2.7**
Total tests failed	1140	516	232	154	96	162	81	23
Fraction of tests failed	46.0%	20.8%	9.4%	6.2%	3.9%	6.5%	3.3%	0.9%

3.0	**3.1**	**3.2**	**3.3**	**3.4**	**3.5**	**3.6**	**3.7**	**3.8**	**3.9**	**3.10**	**3.11**
0	45	47	73	16	170	365	125	87	197	196	540
0.0%	1.8%	1.9%	2.9%	0.6%	6.9%	14.7%	5.0%	3.5%	8.0%	7.9%	21.8%

e.g., `collections.Counter` was introduced in both Python 2.7 and 3.1. In such cases we count both. (3) The results of all these files are first mapped to the Python version it was introduced in, then mapped to the feature's name with the number of times it occurs in the whole project. This mapping is also saved to a file. (4) By summing up all non-unique features per Python version and dividing it by the total number of features detected in this project, we generate the modernity signature.

We automated the calculation of this modernity signature by building a tool on top of `Vermin` that executes these steps called `Pyternity` — a portmanteau of "Python" and "modernity". The implementation of `Pyternity`, the results of our experiments, all raw data and all graphs presented here can be found on https://github.com/grammarware/modernity-python, openly available under the MIT license. In the `README.md` file one can find more information on which commands to run, to reproduce the experiment. Furthermore, to increase reproducibility `Pyternity` provides an extensive command line interface, allowing the tool to be run on any given PyPI project.

Validation of vermin. Since `Pyternity` is built on top of `Vermin`, the latter requires additional investigation. To ensure Vermin's ability to detect new features for each Python release, we validate its feature detection. This involves examining the documentation of the Python standard library and its various changelogs to identify new features. Due to the large number of occurrences of such text in the Python 3 documentation, we automated this process: To find these new features, we parse Python's source documentation using *Sphinx* (https://www.sphinx-doc.org/), a documentation generator used for Python, into *doctrees*. Thus, each doctree represents a documentation page. We then traverse the doctree and look for all `versionadded` and `versionchanged` nodes in the tree. For each of these nodes, we may then automatically generate a test case. Hence, test cases are generated if a new module, function, class, method, parameter, constant, attribute or exception has been added. Repeating this for the considered 20 Python releases from Python 2.0 to Python 3.11 we generate a total of 2,476 test cases. The outcomes of these test cases are displayed in Table 2. The upper table lists the test cases that were generated specifically per Python version whereas the lower table presents the count and percentage of the overall test cases that did not pass for each version. The test failures were reported to `Vermin` Admiraal (2023b) and are being addressed by the package developer.

Table 2 reveals that most tests fail for the oldest and newest Python versions, respectively. This is to be expected, since `Vermin`'s latest update might lag behind the most recent Python release. Additionally, the number of tests for Python 2.0 is quite low, implying that a failed test case counts more relatively towards the total failed %. This is due to `Vermin` not offering support for Python versions prior to 2.0. A notable gap is for the coverage of Python 3.0. This is due to the lacking documentation not mentioning major version changes.

Additionally, we highlight two inherent issues of the Python documentation itself: (1) There is no consistency to determine whether a specific `versionchanged` node is describing a new parameter or a behavioural change. This is not enforced by the `devguide` of Python (Python Software Foundation, 2023b). (2) The Python documentation contains errors. For example, in the `_wingreg` module of Python 2.7.18, ``` ``New in version 2.7.'' ``` is wrongly indented for the functions `CreateKeyEx` and `DeleteKeyEx`. Consequently, there will be no test cases generated for these functions.

5.3. Corpus construction

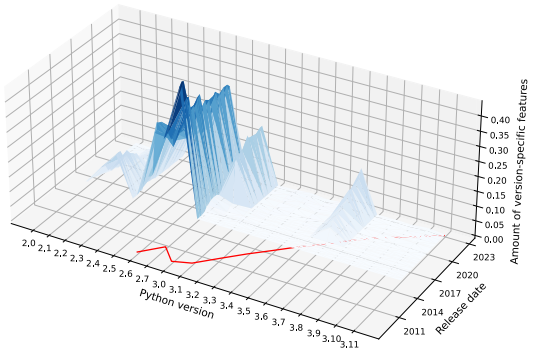
The experiment is run on the 49 most downloaded PyPI projects as of 01-01-2023, gathered from the *Top PyPI Packages* list (van Kemenade et al., 2023). A summary of the corpus can be found in Table 3. All data is retrieved from PyPI's JSON API (Python Software Foundation, 2023a). Note that we calculate the signature only for all minor versions of these projects, *not* micro versions. While we only discuss six handpicked projects here, the full dataset and all graphs can be found on the GitHub repository https://github.com/grammarware/modernity-python.

5.4. Results and discussion

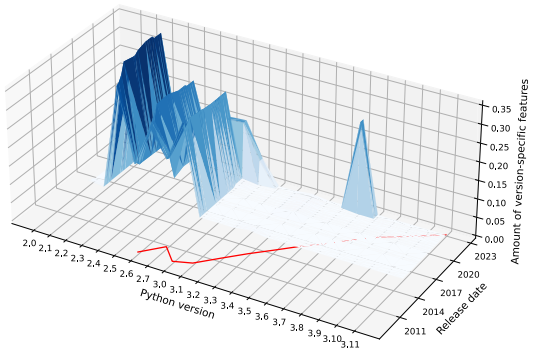
All experiments and tests listed in this research are run with `Pyternity` and `Vermin` using Python 3.11.0 on a Windows 10 (21H2) machine with 16 GB of RAM and an 8-threaded Intel® Core™ i7-7700HQ processor. When running the program on projects introduced later then Python 3.11.0, one should use the latest Python and `Vermin` version to detect new features. Running `Pyternity` on the aforementioned environment and data took roughly two hours. This comprises 146,362 Python files (1,585 MiB) spanning 1,570 releases. The signatures of the 49 projects were combined into one graph, cf. Fig. 7. Conversely, Fig. 6 presents six projects that were handpicked to highlight and discuss trends and exceptions in these trends.

Evidently, many Python 2 features are still used nowadays, even though this differs per minor version and per project. After 2017, we do see an increasing use of Python 3.5 features. Besides the introduction of the `typing` module and *Additional Unpacking Generalisations*, the most common feature detected for this Python version are the co-routines `async` and `await`. For instance, this is clearly visible for *google-api-core* in Fig. 6(c). This observation is also supported by its changelog which mentions this AsyncIO integration (Google, 2023).

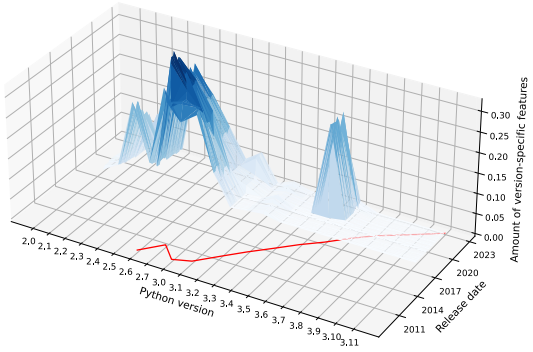
In all graphs the red line shows the minor Python releases through time. Note the drop after version 2.7, since Python 3.0 and 3.1 released before Python 2.7. Naturally, it should not be possible for any data to be shown to the right of this line, as this would imply the usage of features of a Python version that has not been released yet. That is, the release date of the minimum Python version required to run



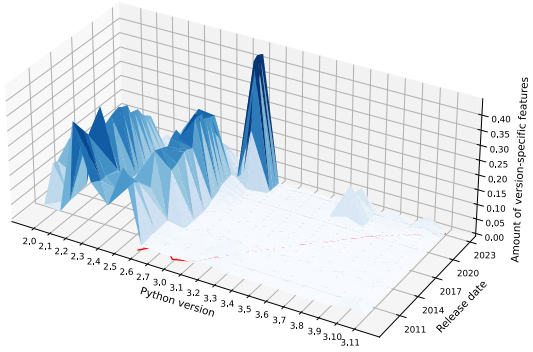
(a) Attrs



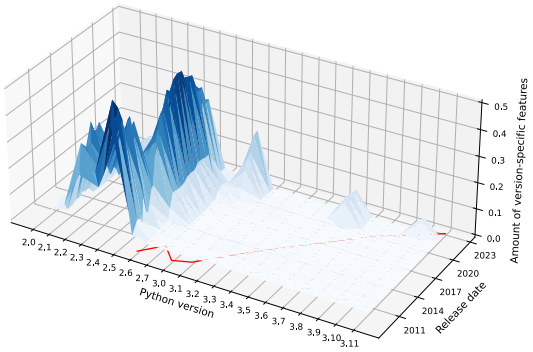
(b) Boto3



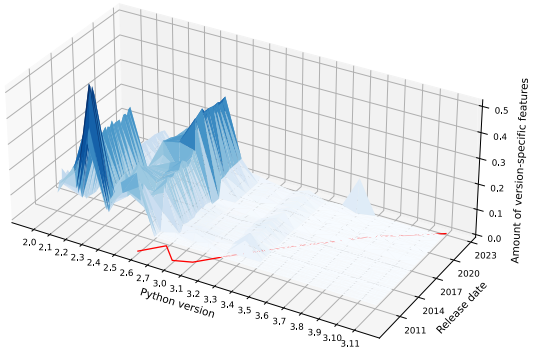
(c) google-api-core



(d) NumPy



(e) pandas



(f) Requests

Fig. 6. Modernity signatures of popular Python projects. The red line indicates the release of a Python version versus the release of the project version. A modernity signature on the right side of this line implies that a feature is used before it was released. This can be the case if names chosen by project developers coincide with official names used in Python version releases. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Table 3
Corpus of the Python case study. The corpus is comprised of the 49 most popular packages according to https://pypi.org/ on 01-01-2023.

Project	Home page	Versions
aiobotocore	https://github.com/aio-libs/aiobotocore	21
attrs	https://www.attrs.org/en/stable/	25
awscli	https://aws.amazon.com/cli/	38
boto3	https://github.com/boto/boto3	27
botocore	https://github.com/boto/botocore	135
cachetools	https://github.com/tkem/cachetools/	21
certifi	https://github.com/certifi/python-certifi	1
cffi	https://cffi.readthedocs.io	24
charset-normalizer	https://github.com/Ousret/charset_normalizer	10
click	https://palletsprojects.com/p/click/	36
colorama	https://github.com/tartley/colorama	4
cryptography	https://github.com/pyca/cryptography	38
docutils	https://docutils.sourceforge.io/	16
fsspec	https://github.com/fsspec/filesystem_spec	23
google-api-core	https://github.com/googleapis/python-api-core	48
google-auth	https://github.com/googleapis/google-auth-library-python	62
google-cloud-bigquery	https://github.com/googleapis/python-bigquery	79
googleapis-common-protos	https://github.com/googleapis/python-api-common-protos	15
grpcio-status	https://grpc.io/	31
idna	https://pypi.org/project/idna/	26
importlib-metadata	https://github.com/python/importlib_metadata	62
jinja2	https://palletsprojects.com/p/jinja/	14
jmespath	https://github.com/jmespath/jmespath.py	11
markupsafe	https://palletsprojects.com/p/markupsafe/	18
numpy	https://numpy.org/	21
oauthlib	https://github.com/oauthlib/oauthlib	13
packaging	https://pypi.org/project/packaging/	40
pandas	https://github.com/wesm/pandas2	31
pip	https://pip.pypa.io/en/stable/	39
protobuf	https://protobuf.dev/	22
pyarrow	https://arrow.apache.org/	18
pyparser	https://keep.imfreedom.org/grim/pyparser	20
pyjwt	https://github.com/jpadilla/pyjwt	17
pyparsing	https://github.com/pyparsing/pyparsing/	7
python-dateutil	https://github.com/dateutil/dateutil	10
pytz	https://pythonhosted.org/pytz/	44
pyyaml	https://pyyaml.org/	9
requests	https://requests.readthedocs.io/	43
requests-oauthlib	https://github.com/requests/requests-oauthlib	11
rsa	https://stuvel.eu/software/rsa/	18
s3fs	https://github.com/PyFilesystem/s3fs	22
s3transfer	https://github.com/boto/s3transfer	6
setuptools	https://github.com/pypa/setuptools	266
six	https://github.com/benjaminp/six	18
typing-extensions	https://pypi.org/project/typing-extensions/	6
urllib3	https://urllib3.readthedocs.io/en/stable/	28
wheel	https://github.com/pypa/wheel	37
wrapt	https://github.com/GrahamDumpleton/wrapt	15
zipp	https://github.com/jaraco/zipp	24

this project's release should not exceed the release date of the project itself. However, we do see some data for Python 3.2 - 3.4 before its release. The incorrect peak at 3.3 is caused by *urllib3* and *requests* (Fig. 6(f)). Vermin detects a `TimeoutError` – a Python 3.3 feature. Closer inspection of the source code of affected projects, reveals that the developers defined a `TimeoutError` class themselves. Other such cases are *typing_extensions* and *setuptools*. However, when looking at the code, we do find an if-clause surrounding the used feature, cf. Fig. 8. This highlights the limitation of only using a decorated abstract syntax tree (AST) for analysis.

The peak at version 3.6 in Fig. 6(b) for *Boto3* is the introduction of *f-strings* to the codebase. This is also the case for *attrs* (Fig. 6(a)), with the addition of also using *variable annotations* shortly after the release of Python 3.5. Out of the 2,938,792 detected features, the most common features are: (1) the `with`-statement of Python 2.5, detected 528,769 times in total, (2) the *function decorator* of Python 2.4 used 353,749 times, and (3) *byte strings* of Python 2.6 used 224,301 times.

Besides that we can detect a variety of different smaller errors, and there are many edge cases `Vermin` does not handle correctly. One such curiosity is the following: There are 1.051 Python files where `Vermin` is in conflict; it is not able to tell which minimum Python version is required. It detects Python-2-only features and Python-3-only features in the same file. Naturally, this is not possible assuming that the source code is valid. Additionally, some Python files threw errors during `Vermin`'s feature detection. A `RecursionError` occurred for versions 0.2 - 2.0 of *idna*, and a `TypeError` occurred for *pandas* 1.5.0.

5.5. Conclusions for python

We built the `Pyternity` tool for calculating the modernity signature of Python projects. It relies on the existing tool `Vermin` to detect version specific features. We use this information to construct the n-tuple signature, normalised by the total number of features detected in a project. Some trends were detected by calculating this signature for popular Python projects.

Despite some shortcomings of `Vermin` Admiraal (2023b), we were able to gather useful insights into some of the analysed projects:

(1) Even though no Python version-specific feature should be used before the release of the respective Python version, we saw some evidence that this happened regardless. While this is mostly

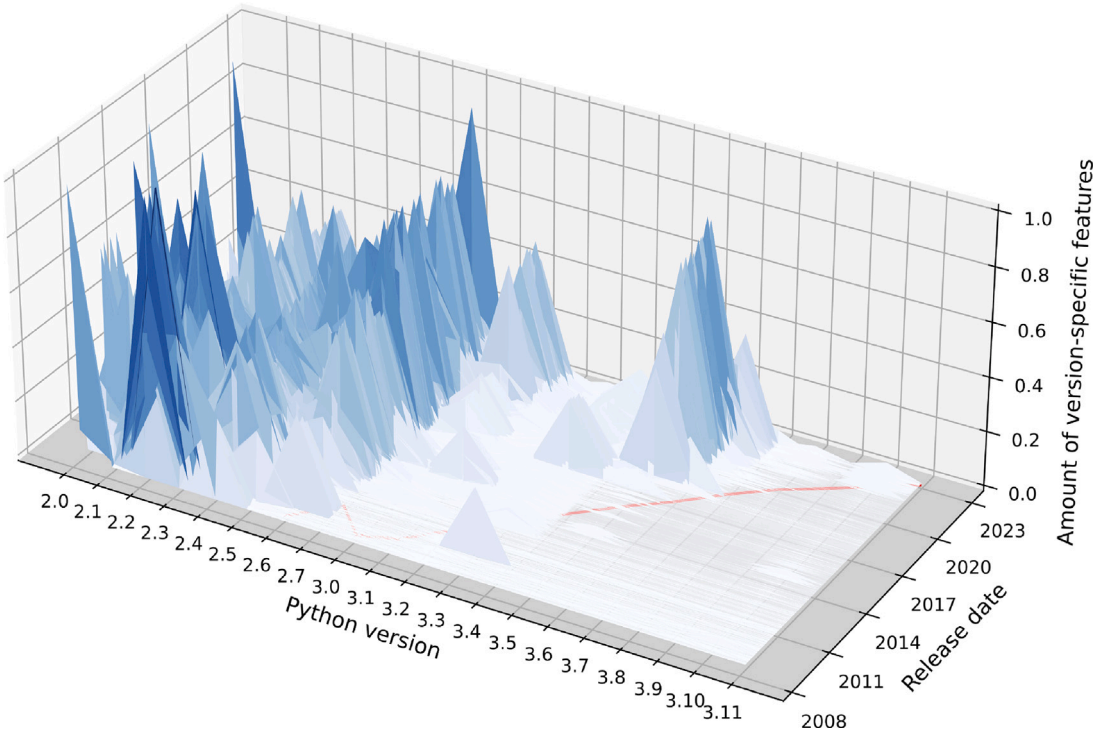


Fig. 7. All modernity signatures of the 49 Python projects combined.

```
if hasattr(typing, 'Required'):
    Required = typing.Required
    NotRequired = typing.NotRequired
elif sys.version_info[:2] >= (3, 9):
    ...
```

Fig. 8. Excerpt of source code of `src/typing_extensions.py` in `typing_extensions` 4.4.0.

due to coinciding naming conventions, modernity signatures first raised a flag to detect this. It could also be explained by the use of intermediate developer versions of the compiler.

(2) The features of some Python versions get adapted quicker and more frequently. A very prominent example is given by Python 3.5 and the newly introduces `await` and `async` syntax which have the capacity to replace much more complex boilerplate-ridden designs. These features cause a surge of this specific version as can be seen in Fig. 7.

(3) Evidently, the vast majority of backwards-compatible features first introduced in Python versions 2.X are still in use today.

Lastly, our analysis made clear that the analysis of the constructed AST should only be the first step when using modernity signatures to analyse a codebase. It should always be complemented with more in-depth investigations of anomalies.

6. On the effect of normalisation techniques

Both case studies employ the use of normalisation in transforming raw modernity signature values into a format that facilitates effective comparison (Vafaei et al., 2022). There is no inherent reason why one technique would work in PHP only and the other would not — They can be used interchangeably, albeit adjusted to the respective language. Despite the significance of normalisation in the context of computing the modernity signatures, the impact of the chosen normalisation techniques on the results was not specifically investigated. We address this here.

The focus of this section is to assess how different normalisation techniques affect these modernity signatures. Our findings could be important, as by broadening the scope of normalisation techniques, we might uncover new perspectives and fresh insights on the evolution of code. Using these insights, we can aid developers in project managements tasks, such as checking whether a codebase adheres to certain development standards or best practices. This could allow developers to make informed decisions regarding project development, and in the long run, lead to an improved technical quality of their code (Maruping et al., 2009).

We assess the influence of different normalisation techniques on modernity signatures, and determine if they reveal new perspectives on a project's level of modernity. We aim to comprehend the effect of each possible normalisation technique on modernity signatures. However, it is also essential to understand the value of each normalisation technique in assessing the evolution of a project's codebase. This leads us to an exploration being guided by the following research questions:

RQ1 *What normalisation techniques are applicable for modernity signatures?*

RQ2 *How do different normalisation techniques impact modernity signatures?*

RQ3 *What insights can different normalisation techniques provide into a project's code evolution?*

6.1. Methodology

We outline the research methodology employed to address the research questions from above. It was designed to systematically investigate the effects of different normalisation techniques on the interpretation of modernity signatures.

6.1.1. Identification of applicable normalisation techniques

Case studies from the preceding sections showcased two distinct normalisation techniques on the modernity signatures. The PHP study employed a method in which the signature was scaled by its maximum element, known as "Max normalisation". In contrast, in the Python study it was chosen to normalise by dividing the signatures by the sum of their elements, a method known as "Sum normalisation". Both of these normalisation techniques belong to the linear category, as explained in the study by Camarinha-Matos et al. (2020). Leveraging their results, we now want to expand the scope of analysis to the semi-linear and non-linear methods.

The selection of appropriate techniques also accounted for the inherent statistical properties of the data. Given that modernity signatures comprise counts of features from various language versions, these signatures are non-negative, ordered, and discrete. This understanding guided our exploration of suitable normalisation techniques summarised in Table 4. Note that some of the chosen methods only require maximal and minimal values of the given vector, while others require the mean and standard deviation.

Our goal in this research is to ensure a multifaceted perspective to mitigate potential bias, thereby enhancing the applicability and robustness of our findings (Smith and Noble, 2014).

6.1.2. Implementation of different normalisation techniques

The next phase was to incorporate these identified normalisation techniques into the existing codebases of the case studies (Admiraal et al., 2023; van den Brink et al., 2022b). This required minor adjustments in the initial implementation for computing modernity signatures to include the additional normalisation methods. Both projects already contain plot generation code for visualising modernity signatures over time in their respective GitHub repositories. This code was adapted to incorporate the varying normalisation methods under review.

6.1.3. Analysis of different normalisation techniques

The crucial stage of this investigation involved scrutinising and contrasting the plots generated as a result of each normalisation technique. For the purpose of this study, a sample set of ten projects were handpicked, ensuring a diverse range of initial modernity signatures. For example, if two projects shared a strikingly similar modernity signature based on our observations of Section 4 and Section 5, only one was included in the sample. This was done in order to increase the likelihood that the findings would generalise to new data (Hand, 2006).

For each normalisation method under examination, new plots were created for all projects in the dataset. These newly generated plots were then juxtaposed against plots generated by the unmodified algorithms of the previous sections. The analysis itself was conducted through manual visual examination, involving a thorough assessment of the presence, magnitude and proportions of the peaks in each plot. We define the patterns we used for examinations below:

- Peak *presence* involves identifying any novel peaks that have emerged in the normalised plot in comparison to the original or discerning any significant peaks that have vanished.
- Peak *magnitude* examination involves comparing the peaks' sizes. The objective here is to identify any instances where the peaks have grown larger or become smaller, marking significant differences in the plotted data's feature usage or distribution.
- Peak *proportions* involves comparing how the peaks in the normalised plots stand relative to each other, versus their placement in the original plots. The goal is to see if the layout of the peaks remains similar or shows significant changes. This part of the analysis looks at the evenness and spread of features from different language versions within the project.

As the colour gradient of the plots is directly dependent on the presence and magnitude of peaks, we do not explicitly consider it in our investigation. In short, analysing the peak presence, magnitude, and proportions should give us a well-rounded view of the data distribution within the modernity signatures.

6.2. Experiment

We describe the steps followed in the experimental part of our research. Specifically, Section 6.2.1 describes the setup of the experiment, including the selected dataset and normalisation techniques to be compared; Section 6.2.2 presents the outcomes of our comparisons; and Section 6.2.3 discusses how different normalisation techniques affect the results; finally, in Section 6.2.4, we discuss what our findings mean in the broader context.

6.2.1. Setup

To simplify the investigation, we focus on the Python case study from Section 5 and specifically on the following PyPI projects: `attrs`, `boto3`, `botocore`, `charset-normaliser`, `fsspec`, `google-api-core`, `Jinja2`, `requests`, `urllib3`, and `wheel`.

Setting up our experiment also covered identifying the normalisation techniques which might be applicable in our research. To address that, we have strategically selected normalisation methods from three distinct categories as identified by Camarinha-Matos et al. (2020). These categories include linear, semi-linear and non-linear methods, cf. Table 4. We point out that other representatives of either category exist, and the ones included in our report were solely chosen based on simplicity.

From the linear category, we included the *Max* and *Max-Min* methods. These techniques offer simple, direct scaling of the data, each focusing on different aspects such as relative peak values and data range (Singh and Singh, 2020). We also employed the *Vector* normalisation method from the semi-linear category, which accounts for the direction and magnitude of data vectors, offering a different perspective on the data (Camarinha-Matos et al., 2020). Our investigation also incorporates the *Logarithmic* method from the non-linear category. This method is adept at handling vast differences in data values, providing a detailed view of subtle data variations (Zolfani et al., 2020). Finally, we incorporated the *Z-score* semi-linear normalisation method, as highlighted in the research of Singh and Singh (2020). This method provides a statistical lens to view the data by representing it in terms of standard deviations from the mean.

Thus, we have two linear, two semi-linear and one non-linear normalisation method in our arsenal. It is important to acknowledge that within each category, Camarinha-Matos et al. describe additional normalisation techniques which could potentially apply in our study. However, for our purposes they were excluded in order to minimise the computational time and potential complexity in implementation and interpretation (this quickly adds up since the computation of modernity signatures involves many normalisation steps). In a similar vein, Singh et al. have elaborated on the existence of other categories of normalisation methods, such as decimal, sigmoidal, and tanh-based methods. However, the decimal methods closely resemble linear methods such as the Max and Max-Min, while the sigmoidal methods align more with non-linear methods, such as the Log technique. Finally, tanh-based methods make use of parameters which if not chosen carefully can lead to an improper scaling of the data (Singh and Singh, 2020). Therefore, the decimal, sigmoidal and tanh-based methods are excluded.

Given that all the required formulae for the selected normalisation methods are known and listed in Table 4, it was straightforward to implement them in code. In the GitHub repository (Admiraal et al., 2023), the `main.py` file can be found, which contains the code directly responsible for normalising the modernity signatures.

The implementation of the Max and Max-Min normalisation techniques leverages Python's built-in `max()` and `min()` functions, respectively, to identify the maximum and minimum elements within the signature. For the Z-score technique, the `numpy` library is utilised. Specifically, the `np.mean()` and `np.std()` functions are employed to calculate the mean and standard deviation. For the Log normalisation technique, the math library's `math.log1p()` function was used to calculate the logarithms.

Table 4
Normalisation formulae.

Technique	Category	Formula
Max	Linear	$x_i^{`} = \frac{x_i}{x_{\max}}$
Max-Min	Linear	$x_i^{`} = \frac{x_i - x_{\min}}{x_{\max} - x_{\min}}$
Vector	Semi-Linear	$x_i^{`} = \frac{x_i}{\sqrt{\sum_{i=1}^{m} x_i^2}}$
Z-Score	Semi-Linear	$x_i^{`} = \frac{x_i - \mu}{\sigma}$
Log	Non-Linear	$x_i^{`} = \frac{\ln x_i}{\ln \prod_{m}^{i=1} x_i}$

Table 5
Normalisation effects on the plots.

Method	Max or max-min	Vector or Z-score	Log
Peak increase	10/10	8/10	10/10
Altered proportions	8/10	2/10	10/10
New peaks	0/10	0/10	10/10

6.2.2. Results

Max normalisation. Upon applying Max normalisation, a distinct increase in peak magnitude was observed across all ten projects within the Max normalised plots. Despite this, only eight projects demonstrated alterations in peak proportions. Interestingly, the emergence or vanishing of peaks was not evident in any of the normalised project plots.

Max-min normalisation. The application of Max-Min normalisation produced comparable outcomes to those of Max normalisation. The peak magnitudes and proportions experienced identical changes.

Vector normalisation. Post Vector normalisation, eight out of ten projects exhibited an escalation in the magnitude of most peaks. The alteration of peak proportions was only detected in two projects. No new peaks emerged or disappeared in the Vector normalised project plots.

Z-score normalisation. The results derived from the Z-score normalisation parallelled those from Vector normalisation. The peak magnitudes and proportions in the Z-score normalised plots mirrored the changes observed in the Vector plots.

Log normalisation. Log normalisation noticeably changed all ten project plots in our dataset. This method caused all peak heights to rise and altered their proportions. A unique result of Log normalisation was that new peaks appeared in every plot.

An overview of the normalisation methods and the number of projects in which the peak increase, altered proportions, and emergence of new peaks were observed, is presented in Table 5 .

6.2.3. Discussion

Max normalisation. The effect of Max normalisation can be explained by the nature of the technique itself, which scales each value of the signature by its maximum element, resulting in a normalised signature with values ranging between zero and one (Singh and Singh, 2020). One example of this can be seen in the Max-normalised plot for the *botocore* project, as illustrated in Fig. 10. Comparing this with the original plot in Fig. 9, it becomes evident that the values for the normalised plot at Python version 2.5 remain consistently at the maximum across all release dates. In contrast, the original plot displays a steady increase in values throughout the various release dates. A similar effect is visible in the Max-normalised plot for the *wheel* project, as shown in Fig. 13. Interestingly, around the year 2020, the peak corresponding to the Python version with the highest values transitions from version 2.1 to 2.5. Upon investigating the reason for this transition, it was observed that in the original plot in Fig. 12, the values for Python version 2.1 begin at a peak and then gradually decrease. Conversely, in the Max-normalised plot, the initial peak for Python version 2.1 remains at a maximum, only decreasing when the feature count of Python 2.5 matches it, thereby leading to a noticeable shift in the peak values (see Fig. 14).

On closer examination, it was observed that plots with lower original values underwent a more drastic transformation. This is primarily

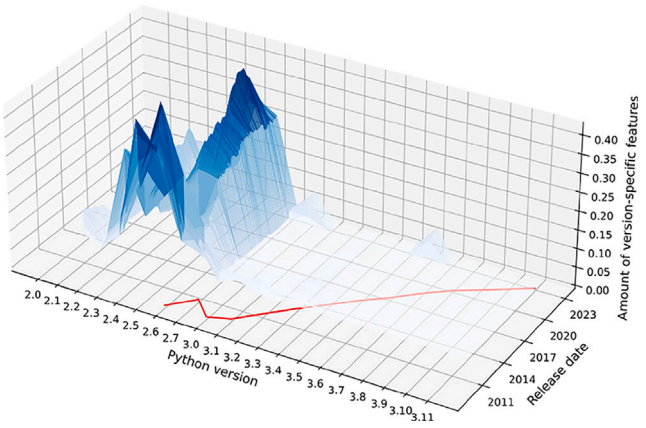


Fig. 9. *botocore* Sum plot (original).

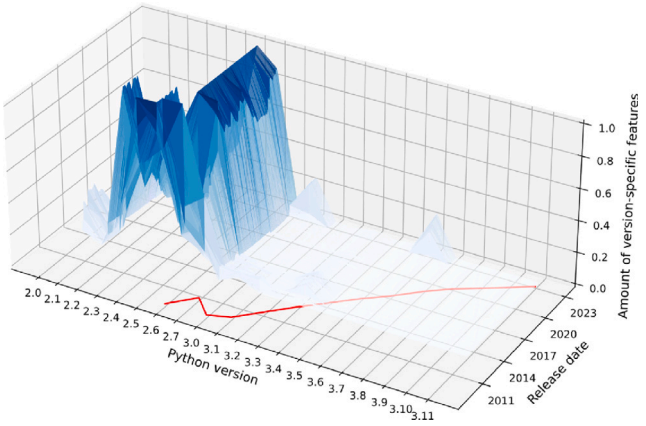


Fig. 10. *botocore* Max plot.

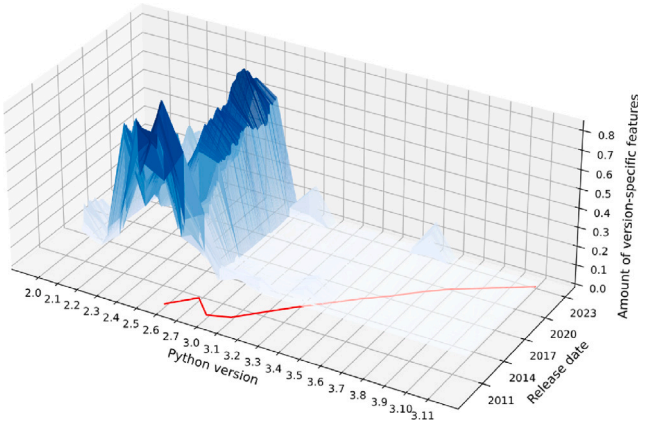


Fig. 11. *botocore* Vector plot.

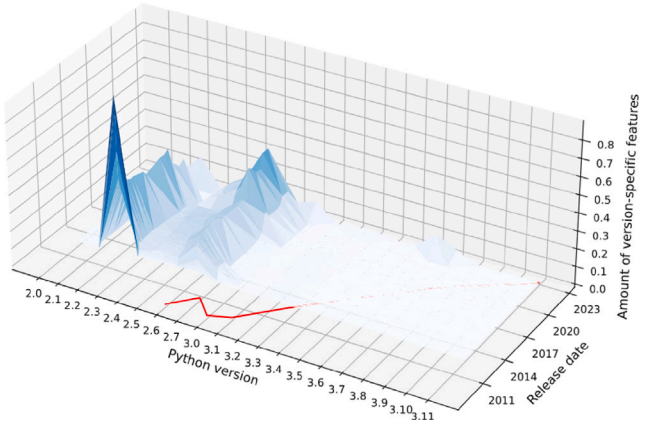


Fig. 12. *wheel* Sum plot (original).

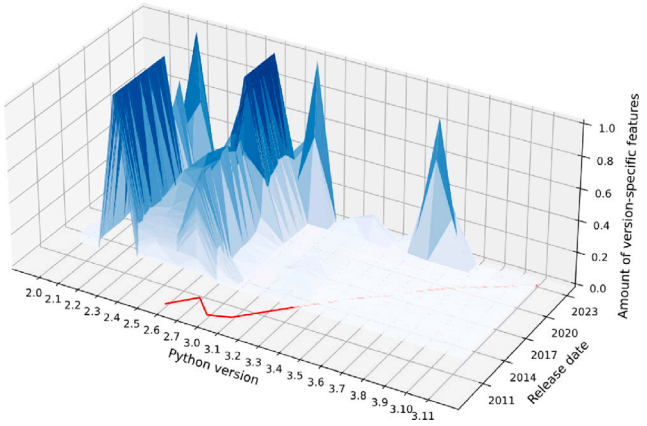


Fig. 13. *wheel* Max plot.

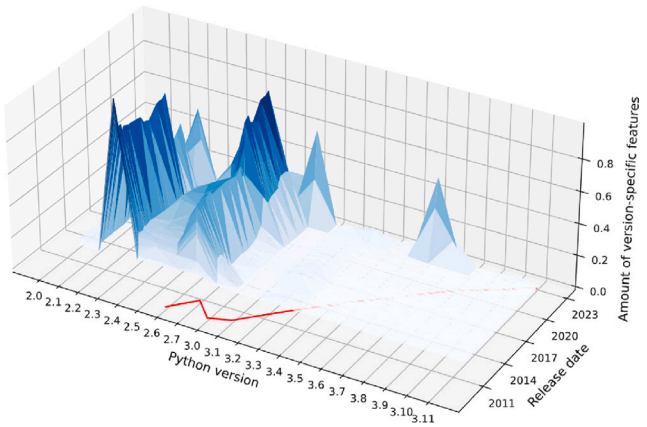


Fig. 14. *wheel* Vector plot.

due to the character of Max normalisation, which scales up the maximum value in any given modernity signature to a peak value, leading to more conspicuous changes in plots with lower initial values.

Ultimately, these findings suggest that Max normalisation effectively accentuates the language version with the most features at a specific release date. By scaling the values so that the maximum becomes one, the highest possible in the normalised scale, it allows a clearer and more immediate identification of which the language version is predominant at each point in time (see Fig. 16).

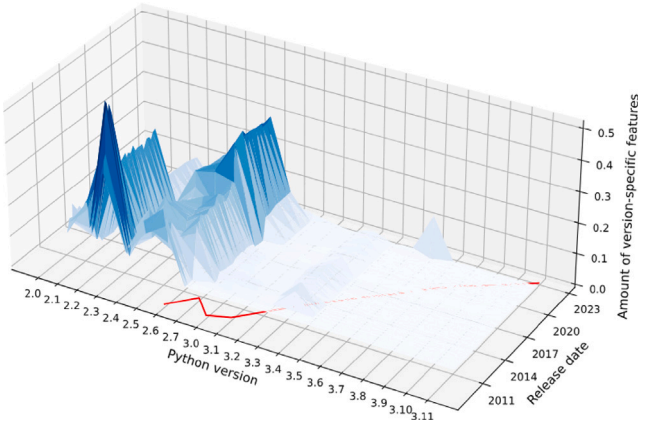


Fig. 15. *requests* Sum plot (original).

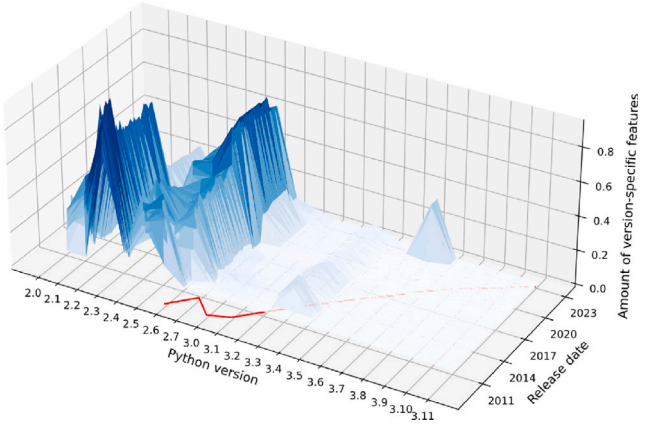


Fig. 16. *requests* Vector plot.

Max-min normalisation. Max-Min normalisation, akin to Max normalisation, produced similar transformations on the plots within our dataset. Both techniques use the same principle: linear rescaling of data to fit a designated range (Singh and Singh, 2020). A key point to note here is that the modernity signatures for all the projects in the dataset contain a value of zero for at least one Python version. Consequently, after applying Max-Min normalisation, the minimum value in the data range becomes zero, essentially aligning the results with those obtained through Max normalisation. Therefore, despite the slightly different methodology, the visual outcomes produced by Max and Max-Min normalisations ended up being literally indistinguishable.

Vector normalisation. The influence of Vector normalisation on the projects within our dataset demonstrated varied outcomes. In total, eight out of the ten projects exhibited an amplified peak magnitude. However, among these, only two projects exhibited alterations in the proportions of the peaks.

The two projects which experienced changes in peak proportions post Vector normalisation were *wheel* and *requests*. The *wheel* project plot (cf. Fig. 12), which had earlier exhibited significant changes under Max normalisation, also showed a considerable difference in its Vector normalised plot (cf.). Although the peak values corresponding to Python versions 2.1 and 2.5 significantly increased in the Vector normalised plot compared to the original, they did not stay at a constant maximum as was the case in the Max normalised plot. Instead, the peaks displayed a more gradual increase and decrease.

Upon investigating why *wheel* and *requests* were the only projects considerably affected by Vector normalisation, it was discovered that both of these projects displayed similar characteristics in their original

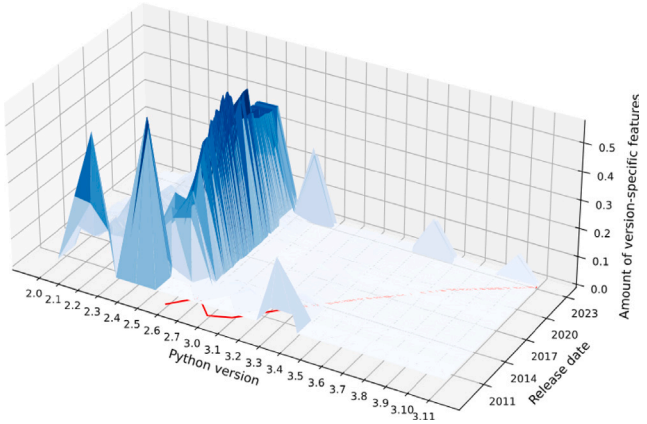


Fig. 17. *urllib3* Sum plot (original).

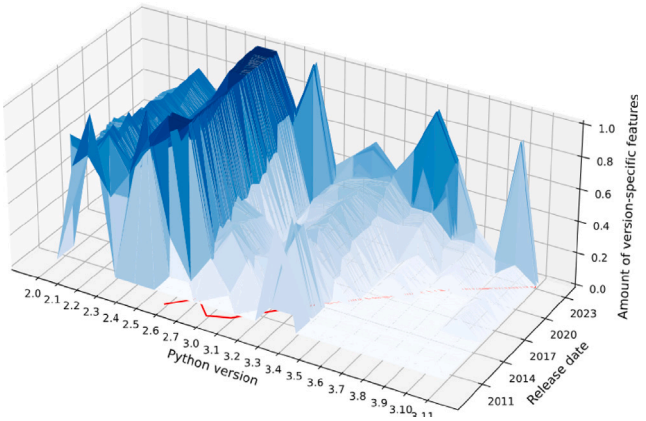


Fig. 18. *urllib3* Sum plot (original).

plots, as shown in Figs. 12 and 15. As previously mentioned in the , for the *wheel* project, the original values for Python version 2.1 steadily decrease over time, whereas those for version 2.5 gradually increase. While it is slightly less apparent, the *requests* project also shows an initial peak followed by a decline for the values at Python version 2.0, simultaneously with a slow increase for Python version 2.5. This results in a shift in the proportions of features from each Python version over time. To put it simply, the share of features from earlier Python versions is decreasing, while that from the newer versions is increasing. This trend reflects a natural evolution as a project's codebase matures over time (Malloy and Power, 2018). The effect of Vector normalisation, however, was not uniformly witnessed across all projects within our dataset. A prime example of this is the *botocore* project, where the Vector normalised plot (cf. Fig. 11) is strikingly similar to its original plot (cf. Fig. 9). This could potentially be attributed to the fact that the contributions from Python version 2.4 already dominate the codebase, with the presence of features from other Python versions being relatively minimal. As a result, in the Vector normalised plot, the influence of Python version 2.4 features escalates steadily, mirroring the pattern observed in the original plot. This could mean that Vector normalisation accentuates shifts in feature contributions from older to newer Python versions over time.

These observations imply that Vector normalisation serves to emphasise transitions in feature contributions from older to more recent Python versions as a project evolves. However, the degree of this effect can be contingent upon the project's original distribution of feature contributions across Python versions. In projects where a single Python version predominantly contributes to the codebase, as in the case of *botocore*, the impact of Vector normalisation may be less discernible. In contrast, for projects that display a shifting balance of feature contributions from different Python versions over time, Vector normalisation can significantly amplify these changes.

Z-score normalisation. Z-score normalisation produced outcomes identical to those induced by Vector normalisation. Both these techniques, despite having different mathematical formulae, effectively rescale the original feature vectors to have standardised properties. Z-score normalisation transforms the data to have a mean of zero and a standard deviation of one (Singh and Singh, 2020). On the other hand, Vector normalisation scales the vectors to have a unit length (Chakraborty and Yeh, 2009). Essentially, both methodologies recalibrate the influence of each Python version's feature contribution to a project to a uniform scale. This results in identical visual patterns in the plots across all projects, reflecting that these normalisation methods provide an exactly similar view of the feature distribution across different Python versions.

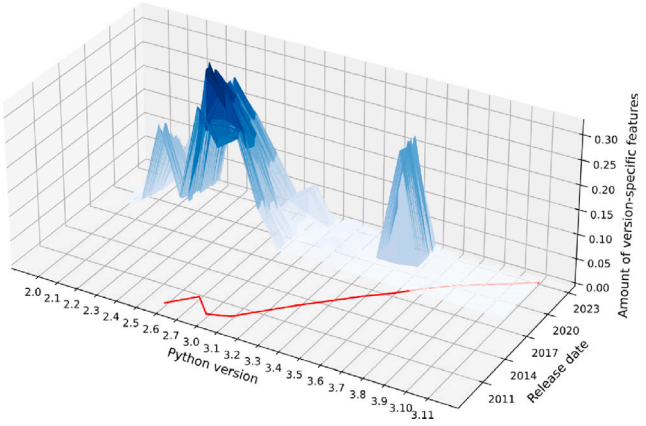


Fig. 19. *google-api-core* Sum plot (original).

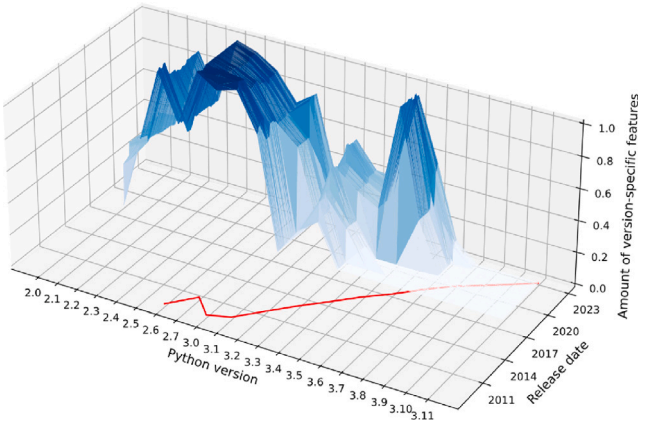


Fig. 20. *google-api-core* Log plot.

Log normalisation. The application of Log normalisation to the project plots in our dataset resulted in an uniform effect on all project plots. This is most evident in the *urllib3* project. In the original plot of this case study (cf. Fig. 17), values after Python version 2.7 are barely noticeable, with minor peaks present at versions 3.1, 3.7, and 3.10. However, the plot's landscape changes significantly when subjected to Log normalisation (cf. Fig. 18).

The slight peaks corresponding to Python versions 3.1, 3.7, and 3.10 become considerably more pronounced in the Log normalised plot, reflecting substantially higher values. Log normalisation unveils the

features associated with other Python versions that were initially indistinguishable in the original plot, contributing additional complexity and detail to the plot. The same trend is notable in the *google-api-core* project. In the original plot (cf. Fig. 19), the visualisation prominently features three peaks linked to Python versions 2.1, 2.3, and 3.5. However, upon the application of Log normalisation (cf. Fig. 20), the sharpness of these peaks is significantly toned down. Simultaneously, the visibility of feature values for Python versions located between these peaks is enhanced, giving a more detailed and comprehensive view of the plot's structure.

Overall, Log normalisation provides a more comprehensive perspective of a project's feature distribution across different Python versions. It uniquely excels at highlighting features associated with Python versions that might have previously remained undetectable, thus offering a nuanced view of the feature landscape.

6.2.4. Implications

Our findings address a key challenge of software evolution, as outlined by Mens et al. namely, the identification and comprehension of various evolution types within codebases (Mens et al., 2005). In Section 6.2.3 we discuss how different normalisation methods give unique insights into how language use in projects changes over time. The techniques from the linear, semi-linear and non-linear categories allow us to understand the code's evolution in different ways. For instance, with the Max and Max-Min methods, we can see the dominant language versions used in a project. The Vector and Z-score methods show us how language use shifts over time, while the Log method lets us see the entire distribution of different features used over time.

To put this in a real scenario, if a development team is using an older version of a language, the Max and Max-Min methods would highlight this by showing a high dominance of features unique to that version. If the team progressively adopts a newer version over time, the Vector and Z-score methods would capture this shift, displaying a gradual transition from the older version's features to the newer one's. Furthermore, the Log method can illustrate how consistently certain features are used across all versions. If a particular feature from an old version continues to be heavily used despite the introduction and adoption of newer versions, this could indicate a significant dependency on that feature in the project's codebase.

Evidently, there is not a single best normalisation technique. Since they all highlight individual aspects of code evolution, we recommend to combine the insights from each method to provide a comprehensive understanding of the code's modernity and its evolution. It is worth stating that the process of normalising the signatures using different techniques to gain new insights is not exclusive to Python. In theory, as long as there is a method of obtaining a modernity signature for a given programming language, the normalisation process is applicable with any of the techniques discussed in this paper. Ultimately, modernity signatures are just sets of numbers, regardless of the programming language. The data produced through our method can also be transformed into meaningful metrics, thereby offering deeper insights into the trajectory of code evolution (Mens and Demeyer, 2001). For instance, using the Max and Max-Min methods the stability of a codebase can be inferred — i.e., how much the dominant language version of a codebase changes over time. For the Vector and Z-score techniques, a transition metric could be deduced which measures the rate at which a codebase transitions from one language version to another. Lastly, for the Log method, a feature persistence metric could measure the consistent use of certain language features across all major and minor language versions. Further integration of these metrics with existing infrastructure, such as version control systems, could enhance the capability to assess code modernity (Mens et al., 2005).

6.3. Threats to validity

The method of visually comparing the plots for obtaining results represents a potential threat to the *internal validity* of our research. While this approach seems to offer intuitive insights into the patterns and effects of the different normalisation techniques, it is also subjective and might introduce bias, as it heavily relies on human interpretation. Consequently, this could potentially affect the overall consistency and reliability of our findings. To address this threat, a more robust and quantitative approach could be integrated that does not rely on human vision. Instead, statistical measures like the mean, standard deviation, skewness and kurtosis could be measured by a computer to allow researchers to better understand the plot transformations each normalisation method brings. Machine learning could also be employed to automatically identify patterns in the normalised data.

The selection of the projects to be studied is another possible threat to the *external validity* of our work. In Section 6.1.3 we explain that the projects were selected such that none of them resemble each other, in order to reduce selection bias. However, this does not mean that our selection of projects adequately represent all the types and shapes of modernity signature plots. Consequently, our findings may not hold true for all kinds of projects in the wild. Including more diverse projects in our research could help determine if our findings apply more broadly. However, given that we rely on human interpretation and visual comparisons, adding more projects to our analysis would notably increase the time complexity of the research.

6.4. Conclusions for normalisation

We address the research questions introduced in the beginning of Section 6.

RQ1 was addressed in Section 6.2.1. We determined that five specific normalisation techniques: Max, Max-Min, Vector, Z-score, and Log normalisation — are applicable to modernity signatures. These techniques span the linear, semi-linear and non-linear categories. While other methods like decimal, sigmoidal and tanh-based methods exist, they closely mirror our chosen techniques or require a complex parameter setup.

RQ2 was addressed in Section 6.2.2 (results) and Section 6.2.3 (interpretation). Our investigation reveals that different normalisation techniques distinctly alter the modernity signatures. Max and Max-Min methods predominantly emphasise the prevailing Python version in each period, while the Vector and Z-score techniques underscore dynamic transitions in feature usage over time. Log normalisation highlights minor variations in feature usage, accentuating even less prominent versions.

RQ3 was addressed in Section 6.2.4. This study reveals that different normalisation methods can uncover various forms of software evolution within a codebase. Moreover, the insights derived from these methods can be translated into metrics. These metrics can then be seamlessly integrated with other systems, such as version control tools, further enriching our understanding of software modernity and the evolutionary trajectory of programming languages.

7. Conclusions and lessons learnt

In this paper, we have introduced the concept of *modernity signatures* and explored it in the context of two programming languages and five normalisation schemes. We can identify the following takeaways from this project:

Lesson 1 – Visualising Co-evolution: Modernity signatures serve as a technique to illustrate the adaptation of newly introduced features of a developing programming language within a codebase that evolves

concurrently. We have conducted experiments on PHP and Python, but the concept lends itself easily to other languages with the same property: parallel co-evolution of the language and the code. This would enable a better understanding of the consistency of normalisation technique impacts across different languages as well. We have ourselves experimented on Java (Güdücü, 2022), which presents an interesting case, as Dyer et al. specify in their work that most of its features see limited use in the current industry (Dyer et al., 2014). Therefore, it would be interesting to study the changes over time in Java projects using different normalisation methods. This could also help determine if these new findings match up with the results of our current research. There are also Vermin-like tools in other languages, such as MSRV (Gribnau et al., 2019) for Rust, which can make modernity calculations simpler and more reliable. The problem of language *version* detection is also a substantially complex one by itself, and does not lend itself that easily to AI solutions (Gerhold et al., 2023) as the problem of *language* identification does (Kennedy van Dam and Zaytsev, 2016).

Lesson 2 – Adoption Patterns: As we have demonstrated, this way of looking at code shows very different pictures for different codebases, indicating diverse strategies for adoption of programming language features. Patterns of programming language feature adoption seem to be language-dependent at least to some extent. It remains to be investigated whether this difference is due to the structure of the languages themselves, or are more ecosystematic and observable due to different trends and tendencies within the communities of language users. In prior research, Farooq and Zaytsev (2021b) have shown that language feature adoption typically follows one of three patterns: (1) a steadily rising one, (2) a one time spike (i.e. a feature gets massively adopted right after release and then massively abandoned) and (3) a "hype curve" pattern with a slowly rising adoption, a peak of disappointment, a trough of disillusionment and an even slower second-generation adoption phase. If these patterns are known per feature, it would be interesting to experiment with some normalisation of the signatures to account for their shape. The patterns also deserve further investigation in the context of becoming somewhat formalised, with proper quantitative justification for this classification, with appropriate thresholds.

Lesson 3 – Normalisation Impact Different normalisation schemes used while computing modernity signatures, have an effect on what elements of the language adoption are accentuated best. For instance, if peaks in language development are pronounced – i.e., it is common knowledge that a particular version of a language has introduced significantly more new features than other versions, – then it is sensible to apply normalisations that emphasise these peaks. If the language evolution is known to be smooth, it means that any peaks are not very pronounced and accidental in nature, which in turn should advice against accentuating normalisations since they would lead to jagged modernity evolution plots with intermittent peaks. On a similar note, projects with more even distribution of adopted features per language version, would profit more from other kinds of normalisations since they tend to amplify otherwise subtle fluctuations. Unfortunately, we failed to identify the single best normalisation techniques that is universally applicable. Perhaps further formalisations and moving to some sort of modernity calculus, could bring further clarity to the issue.

Lesson 4 – Detection Challenges: There are unavoidable obstacles in calculating modernity signatures, and automatic detection algorithms, especially feature-based ones, will always have their limitations. The exact language version used in the codebase depends on careful and detailed analysis of the source code which tends to try to be misleading. For instance, in our previous work on developing an industrial compiler for IBM Assembler targeting .NET in the context of migrating legacy systems to cloud native (Blagodarov et al., 2016; Zaytsev, 2020), we have encountered situations where seemingly the same keyword was used by one customer as an instruction and by another customer as a macro. Detailed investigation with interviews of domain experts showed that it was customary to keep using an older instruction set (e.g., one for HLASM V1R1 with a minimal set barely larger than the BAL version for IBM System/360, but with full support of extended mnemonics) while cherry-picking instructions from later version while defining them locally as macros. This was the system owner's insurance out of the vendor lock-in and their guarantee to not rely on any hardware architecture-specific features (pre-HLASM versions of IBM Assembler during 1980s were diverging), while keeping the options open to retire the macros later when migrating to a newer mainframe. As usual in the conservative world of business-critical legacy software, migrations happened, but retiring did not, and we had to keep our compiler highly configurable with a full definition of what constitutes the instruction set, given at compilation time (Zaytsev, 2020). This means that in some cases the inspection of the code simply does not provide enough information to reliably determine modernity. For example, Python 3.10 introduces asynchronous built-in functions `aiter()` and `anext()`, but calling those functions from the code does not mean its required language level being 3.10+, since these functions can be defined by the developer. Source code analysis always has limits in detecting that: COBOL has its `COPY`, PHP has its `eval()` and Python has its `exec()`.

Lesson 5 – Modernity Applications: While knowing the "minimum supported compiler version" or the exact language dialect, is definitely valuable for maintenance of a codebase (Lämmel and Verhoef, 2001), expanding it to a vector of prominently used language features, which is the modernity signature, has even more applications. Once developers, practitioners and researchers have access to the modernity signature of their project, they can utilise it in many different ways. We give at least three examples where the modernity and age of a codebase can provide useful insights into necessary maintenance. (1) Finding out which parts of the codebase prefer older language features, indicates possible technical debt in the form of legacy code with overreliance on out-of-date idioms. Assuming that language evolution consists of well-designed steps in a consistent direction, contemporary language constructs should increase developers' productivity and correctness. Refactoring that code and modernising it to adopt newer idioms and constructs, can then lead to fewer or less frequent updates and bug-fixes. (2) Security vulnerabilities are often linked to older language (and library) features (Lehman, 1980). Software with a more modern codebase also might already include patched vulnerabilities of security risks. By providing the modernity signature of a project developers may identify better where security risks may lie. (3) As software technology evolves, newer frameworks and libraries may provide better performance than older ones. In many cases new language features are even motivated by opportunities the corresponding compiler can explore to apply more impactful optimisations. A modernity signature could help developers prioritise where to boost performance of their projects best.

Lesson 6 – Language Development Insights: On the language dimension of the modernity evolution plots, there is a lot of information valuable for language developers. Besides a few metalanguages used by language engineers, the user base of a language is usually much larger than the team that designs the language and is responsible for introducing new features to it. Hence, it is quite possible to introduce a feature that either will not get adopted by most developers (while still adding signifiant complexity to the compilers), or will be abandoned after adoption (for instance, for performance or error proneness reasons). Analysing the modernity evolution and proceeding further into deeper analysis of the (weighted) abstract syntax trees that are used to calculate the modernity of a project, one can prioritise what features to test the compiler implementation for, and where to focus support activities on. This could both provide a safety cushion in thoroughly testing heavily used language features, as well as feed discussions about future extensions to the language with concrete data usage. We see this as a big opportunity especially for domain-specific languages.

CRediT authorship contribution statement

Chris Admiraal: Software, Visualization. **Wouter van den Brink:** Software, Visualization. **Marcus Gerhold:** Methodology, Supervision, Writing – original draft, Writing – review & editing. **Vadim Zaytsev:** Methodology, Project administration, Supervision, Writing – original draft, Writing – review & editing. **Cristian Zubcu:** Data curation, Software, Visualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

Admiraal, C., 2023a. Calculating the Modernity of Popular Python Projects (Bachelor's thesis). Universiteit Twente, http://purl.utwente.nl/essays/94375.

Admiraal, C., 2023b. Library features that are not (correctly) detected · Issue #144 · `netromdk/vermin`. https://github.com/netromdk/vermin/issues/144.

Admiraal, C.P., et al., 2023. Pyternity. URL https://github.com/grammarware/modernity-python.

Ajami, S., Woodbridge, Y., Feitelson, D.G., 2019. Syntax, predicates, idioms — What really affects code complexity? Empir. Softw. Eng. 24 (1), 287–328. http://dx.doi.org/10.1007/s10664-018-9628-3.

Alexandru, C.V., Merchante, J.J., Panichella, S., Proksch, S., Gall, H.C., Robles, G., 2018. On the usage of pythonic idioms. In: Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. In: Onward!, ACM, pp. 1–11. http://dx.doi.org/10.1145/3276954.3276960.

Aljedaani, W., Peruma, A., Aljohani, A., Alotaibi, M., Mkaouer, M.W., Ouni, A., Newman, C.D., Ghallab, A., Ludi, S., 2021. Test smell detection tools: A systematic mapping study. In: Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering. In: EASE, ACM, New York, NY, USA, pp. 170–180. http://dx.doi.org/10.1145/3463274.3463335.

Azeem, M.I., Palomba, F., Shi, L., Wang, Q., 2019. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. Inf. Softw. Technol. 108, 115–138. http://dx.doi.org/10.1016/j.infsof.2018.12.009.

Banyard, G.P., 2021. PHP RFC: Pure Intersection Types. Tech. Rep., The PHP Group, The PHP.net wiki, URL https://wiki.php.net/rfc/pure-intersection-types.

Benureau, F.C.Y., Rougier, N.P., 2018. Re-run, repeat, reproduce, reuse, replicate: Transforming code into scientific contributions. Front. Neuroinform. 11, http://dx.doi.org/10.3389/fninf.2017.00069.

Blagodarov, V., Jaradin, Y., Zaytsev, V., 2016. Tool demo: Raincode assembler compiler. In: van der Storm, T., Balland, E., Varró, D. (Eds.), Proceedings of the Ninth International Conference on Software Language Engineering. SLE, pp. 221–225. http://dx.doi.org/10.1145/2997364.2997387.

Bucher, B., Moisset, D.F., Kohn, T., Levkivskyi, I., van Rossum, G., Talin, 2020. PEP 622 – structural pattern matching. https://peps.python.org/pep-0622/.

C.A. Technologies, 2016. 5 ways DevOps practices boost innovation on the mainframe. CS 200-227965, https://docs.broadcom.com/doc/5-ways-devops-practices-boost-innovation-on-the-mainframe.

Camarinha-Matos, L.M., Farhadi, N., Lopes, F., Pereira, H. (Eds.), 2020. Technological Innovation for Life Improvement. Springer International Publishing, http://dx.doi.org/10.1007/978-3-030-45124-0.

Chakraborty, S., Yeh, C.-H., 2009. A simulation comparison of normalization procedures for TOPSIS. In: International Conference on Computers & Industrial Engineering. pp. 1815–1820. http://dx.doi.org/10.1109/ICCIE.2009.5223811.

Challapalli, S.S.N., Kaushik, P., Suman, S., Shivahare, B.D., Bibhu, V., Gupta, A.D., 2021. Web development and performance comparison of web development technologies in node.js and python. In: Proceedings of the International Conference on Technological Advancements and Innovations. ICTAI, pp. 303–307. http://dx.doi.org/10.1109/ICTAI53825.2021.9673464.

Chen, Z., Chen, L., Ma, W., Xu, B., 2016. Detecting code smells in python programs. In: 2016 International Conference on Software Analysis, Testing and Evolution. SATE, IEEE, Kunming, China, pp. 18–23. http://dx.doi.org/10.1109/SATE.2016.10.

dours, 2021. Do you have plans to support Python 3.10 (for example, assignment expressions)? · Issue #2462 · `antlr/grammars-v4`. https://github.com/antlr/grammars-v4/issues/2462.

Dubois, E., Galindo, A.N.n., Dayon, L., Cominetti, O., 2020. Comparison of normalization methods in clinical research applications of mass spectrometry-based proteomics. In: IEEE Conference on Computational Intelligence in Bioinformatics and Computational Biology. CIBCB, pp. 1–10. http://dx.doi.org/10.1109/CIBCB48159.2020.9277702.

Dyer, R., Rajan, H., Nguyen, H.A., Nguyen, T.N., 2014. Mining billions of AST nodes to study actual and potential usage of Java language features. In: Proceedings of the 36th International Conference on Software Engineering. In: ICSE, ACM, New York, NY, USA, pp. 779–790. http://dx.doi.org/10.1145/2568225.2568295.

Farooq, A., Zaytsev, V., 2021a. F-strings and string formatting. https://slimshadyiam.github.io/ZenYourPython/#/fstrings.

Farooq, A., Zaytsev, V., 2021b. There is more than one way to zen your python. In: Visser, E., Kolovos, D., Söderberg, E. (Eds.), Proceedings of the 14th International Conference on Software Language Engineering. ACM, SLE, pp. 68–82. http://dx.doi.org/10.1145/3486608.3486909.

Favre, J.-M., Nguyen, T., 2004. Towards a megamodel to model software evolution through transformations. In: Heckel, R., Mens, T. (Eds.), Proceedings of the Workshop on Software Evolution Through Transformations: Model-Based Vs. Implementation-Level Solutions, SETra. In: ENTCS, (no. 3), Elsevier, pp. 59–74. http://dx.doi.org/10.1016/j.entcs.2004.08.034.

Fontana, F., Zanoni, M., Marino, A., Mäntylä, M., 2013. Code smell detection: Towards a machine learning-based approach. In: Proceedings of the 29th International Conference on Software Maintenance. ICSM, IEEE, pp. 396–399. http://dx.doi.org/10.1109/ICSM.2013.56.

Forcier, J., Bissex, P., Chun, W.J., 2008. Python Web Development with Django. Addison-Wesley Professional.

Garfield, L., Tovilo, I., 2020. PHP RFC: Enumerations. Tech. Rep., The PHP Group, The PHP.net wiki, URL https://wiki.php.net/rfc/enumerations.

Gerhold, M., Solovyeva, L., Zaytsev, V., 2023. Leveraging deep learning for Python version identification. In: Madeiral, F., Rastogi, A., Wessel, M. (Eds.), Proceedings of the 22nd Belgium-Netherlands Software Evolution Workshop. BENEVOL, CEUR.

Google, 2023. Changelog — `google-api-core` documentation. https://googleapis.dev/python/google-api-core/latest/changelog.html#id118.

Gribnau, M., et al., 2019. MSRV: Minimum supported rust version. https://foresterre.github.io/cargo-msrv/.

Grinberg, M., 2018. Flask Web Development: Developing Web Applications with Python. O'Reilly Media, Inc..

Güdücü, B., 2022. Weighted abstract syntax trees for program comprehension in Java. Universiteit Twente, http://purl.utwente.nl/essays/91735.

Han, D., Ragkhitwetsagul, C., Krinke, J., Paixão, M., Rosa, G., 2020. Does code review really remove coding convention violations? In: Proceedings of the 20th International Working Conference on Source Code Analysis and Manipulation. SCAM, IEEE, pp. 43–53. http://dx.doi.org/10.1109/SCAM51674.2020.00010.

Hand, D.J., 2006. Classifier technology and the illusion of progress. Statist. Sci. 21 (1), 1–14. http://dx.doi.org/10.1214/088342306000000060.

Hauzar, D., Kofron, J., 2015. Framework for static analysis of PHP applications. In: Proceedings of the 29th European Conference on Object-Oriented Programming. In: Leibniz International Proceedings in Informatics, vol. 37, Schloss Dagstuhl — Leibniz-Zentrum für Informatik, ECOOP, pp. 689–711. http://dx.doi.org/10.4230/LIPIcs.ECOOP.2015.689.

Heinl, M.P., Giehl, A., Graif, L., 2020. AntiPatterns regarding the application of cryptographic primitives by the example of ransomware. In: Volkamer, M., Wressnegger, C. (Eds.), Proceedings of the 15th International Conference on Availability, Reliability and Security. ARES, ACM, pp. 64:1–64:10. http://dx.doi.org/10.1145/3407023.3409182.

Hills, M., Klint, P., 2014. PHP AiR: Analyzing PHP systems with rascal. In: Demeyer, S., Binkley, D., Ricca, F. (Eds.), Proceedings of the Software Evolution Week: Conference on Software Maintenance, Reengineering, and Reverse Engineering. IEEE Computer Society, CSMR-WCRE, pp. 454–457. http://dx.doi.org/10.1109/CSMR-WCRE.2014.6747217.

Hills, M., Klint, P., Vinju, J.J., 2017. Enabling PHP software engineering research in rascal. Sci. Comput. Programm. 2 (134), 37–46. http://dx.doi.org/10.1016/J.SCICO.2016.05.003.

Idris, I., 2014. Python Data Analysis. Packt Publishing Ltd.

JetBrains, 2022. Code inspections | PhpStorm. https://www.jetbrains.com/help/phpstorm/code-inspection.html.

Jones, C., 1995. Patterns of large software systems: Failure and success. Computer 28 (3), 86–87. http://dx.doi.org/10.1109/2.366170.

Karaivanov, S., Raychev, V., Vechev, M.T., 2014. Phrase-based statistical translation of programming languages. In: Black, A.P., Krishnamurthi, S., Bruegge, B., Ruskiewicz, J.N. (Eds.), Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software. ACM, Onward!, pp. 173–184. http://dx.doi.org/10.1145/2661136.2661148.

Karunaratne, A., 2020. Php's resource to object transformation. PHP:Watch, https://php.watch/articles/resource-object.

Kennedy van Dam, J., Zaytsev, V., 2016. Software language identification with natural language classifiers. In: Inoue, K., Kamei, Y., Lanza, M., Yoshida, N. (Eds.), Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering: The Early Research Achievements Track. IEEE, SANER ERA, pp. 624–628. http://dx.doi.org/10.1109/SANER.2016.92.

Kim, D.J., 2020. An empirical study on the evolution of test smell. In: Companion Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering. In: ICSE-Companion, pp. 149–151. http://dx.doi.org/10.1145/3377812.3382176.

Kim, D.J., Chen, T.-H., Yang, J., 2021. The secret life of test smells — An empirical study on test smell evolution and maintenance. Empir. Softw. Eng. 26 (5), 100. http://dx.doi.org/10.1007/s10664-021-09969-1.

Kohn, T., van Rossum, G., 2020. PEP 635 – structural pattern matching: Motivation and rationale. https://peps.python.org/pep-0635.

Kristensen, M., 2018. Vermin. https://pypi.org/project/vermin.

Lämmel, R., 2016. Coupled software transformations revisited. In: van der Storm, T., Balland, E., Varró, D. (Eds.), Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering. ACM, pp. 239–252. http://dx.doi.org/10.1145/2997364.2997366.

Lämmel, R., 2018. Software Languages: Syntax, Semantics, and Metaprogramming. Springer, http://dx.doi.org/10.1007/978-3-319-90800-7.

Lämmel, R., Verhoef, C., 2001. Cracking the 500-language problem. IEEE Software 18 (6), 78–88. http://dx.doi.org/10.1109/52.965809.

Lämmel, R., Zaytsev, V., 2009. An introduction to grammar convergence. In: Leuschel, M., Wehrheim, H. (Eds.), Proceedings of the Seventh International Conference on Integrated Formal Methods. In: LNCS, vol. 5423, Springer, iFM, pp. 246–260. http://dx.doi.org/10.1007/978-3-642-00255-7_17.

Laville, L., 2022. PHP compatinfo home page. https://llaville.github.io/php-compatinfo/6.x/.

Lehman, M.M., 1980. On understanding laws, evolution, and conservation in the large-program life cycle. J. Syst. Software 1, 213–221. http://dx.doi.org/10.1016/0164-1212(79)90022-0.

Liu, H., Jin, J., Xu, Z., Zou, Y., Bu, Y., Zhang, L., 2021. Deep learning based code smell detection. IEEE Trans. Softw. Eng. 47 (9), 1811–1837. http://dx.doi.org/10.1109/TSE.2019.2936376.

Malloy, B.A., Power, J.F., 2018. An empirical analysis of the transition from Python 2 to Python 3. Empir. Softw. Eng. 24 (2), 751–778. http://dx.doi.org/10.1007/s10664-018-9637-2.

Maruping, L.M., Zhang, X., Venkatesh, V., 2009. Role of collective ownership and coding standards in coordinating expertise in software project teams. Eur. J. Inf. Syst. 18 (4), 355–371. http://dx.doi.org/10.1057/ejis.2009.24.

Mateos, C., Zunino, A., Flores, A., Misra, S., 2019. COBOL systems migration to SOA: Assessing antipatterns and complexity. Inf. Technol. Control 48 (1), 71–89. http://dx.doi.org/10.5755/j01.itc.48.1.21566.

McKinney, W., 2012. Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython. O'Reilly Media, Inc..

Mens, T., Demeyer, S., 2001. Future trends in software evolution metrics. In: Proceedings of the 4th International Workshop on Principles of Software Evolution. IWPSE, ACM, New York, NY, USA, pp. 83–86. http://dx.doi.org/10.1145/602461.602476.

Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., Jazayeri, M., 2005. Challenges in software evolution. In: Eighth International Workshop on Principles of Software Evolution. IWPSE'05, pp. 13–22. http://dx.doi.org/10.1109/IWPSE.2005.7.

Merlo, E., Letarte, D., Antoniol, G., 2007. Automated protection of PHP applications against SQL-injection attacks. In: Krikhaar, R.L., Verhoef, C., Di Lucca, G.A. (Eds.), Proceedings of the 11th European Conference on Software Maintenance and Reengineering. IEEE Computer Society, CSMR, pp. 191–202. http://dx.doi.org/10.1109/CSMR.2007.16.

Muscat, I., 2016. Web vulnerabilities: Identifying patterns and remedies. Netw. Secur. 2016 (2), 5–10. http://dx.doi.org/10.1016/S1353-4858(16)30016-2.

Nelli, F., 2015. Python Data Analytics: Data Analysis and Science using PANDAs, MatPlotLib and the Python Programming Language. A Press.

Otwell, T., 2011. Laravel: The PHP framework for web artisans. https://laravel.com.

Papagiannis, I., Migliavacca, M., Pietzuch, P.R., 2011. PHP aspis: Using partial taint tracking to protect against injection attacks. In: Fox, A. (Ed.), Proceedings of the Second USENIX Conference on Web Application Development. USENIX Association, WebApps, pp. 1–12, URL https://www.usenix.org/legacy/events/webapps11/tech/final_files/Papagiannis.pdf.

Peng, Y., Zhang, Y., Hu, M., 2021. An empirical study for common language features used in python projects. In: IEEE International Conference on Software Analysis, Evolution and Reengineering. SANER, IEEE, pp. 24–35. http://dx.doi.org/10.1109/SANER50967.2021.00012.

Python, 2008. What's new in Python 3.0. https://www.python.org/download/releases/3.0/whatsnew.

Python, 2021. What's new in Python 3.10. https://docs.python.org/3/whatsnew/3.10.html.

Python, 2022. `Ast` — Abstract syntax trees.

Python Software Foundation, 2023a. Pypi JSON API. https://warehouse.pypa.io/api-reference/json.html.

Python Software Foundation, 2023b. Python developer's guide: reStructuredText markup. https://devguide.python.org/documentation/markup/#paragraph-level-markup.

Raschka, S., Mirjalili, V., 2019. Python Machine Learning: Machine Learning and Deep Learning with Python, `Scikit-Learn` and TensorFlow 2. Packt Publishing Ltd.

Roose, B., 2022. PHP version stats: July, 2022. https://stitcher.io/blog/php-version-stats-july-2022.

Sakulniwat, T., Kula, R.G., Ragkhitwetsagul, C., Choetkiertikul, M., Sunetnanta, T., Wang, D., Ishio, T., Matsumoto, K., 2019. Visualizing the usage of pythonic idioms over time: A case study of the `with open` idiom. In: Proceedings of the 10th International Workshop on Empirical Software Engineering in Practice. IWESEP, pp. 43–435. http://dx.doi.org/10.1109/IWESEP49350.2019.00016.

Sharma, V.S., Anwer, S., 2014. Performance antipatterns: Detection and evaluation of their effects in the cloud. In: Proceedings of the International Conference on Services Computing. SCC, IEEE Computer Society, pp. 758–765. http://dx.doi.org/10.1109/SCC.2014.103.

Sharma, T., Spinellis, D., 2018. A survey on software smells. J. Syst. Softw. 138, 158–173. http://dx.doi.org/10.1016/j.jss.2017.12.034.

Singh, D., Singh, B., 2020. Investigating the impact of data normalization on classification performance. Appl. Soft Comput. 97, 105524. http://dx.doi.org/10.1016/j.asoc.2019.105524.

Smith, J., Noble, H., 2014. Bias in research. Evidence-Based Nursing 17 (4), 100–101.

Taskaya, B., 2020. Soft keywords and how to implement them · issue #138 · `davidhalter/parso`. https://github.com/davidhalter/parso/issues/138.

The P.H.P. Documentation Group, 2022. PHP: History of PHP. https://www.php.net/manual/en/history.php.

The PHP group, 2020. PHP 8.0 released. https://www.php.net/releases/8.0/.

TIOBE, 2023. The TIOBE programming community index. https://www.tiobe.com/tiobe-index/.

Vafaei, N., Ribeiro, R.A., Camarinha-Matos, L.M., 2022. Assessing normalization techniques for simple additive weighting method. Procedia Comput. Sci. 199, 1229–1236. http://dx.doi.org/10.1016/j.procs.2022.01.156, The 8th International Conference on Information Technology and Quantitative Management (ITQM 2020 & 2021): Developing Global Digital Economy after COVID-19.

van den Brink, W., 2022. Weighed and Found Legacy: Modernity Signatures for PHP Systems Using Static Analysis (Bachelor's thesis). Universiteit Twente, http://purl.utwente.nl/essays/91794.

van den Brink, W., Gerhold, M., Zaytsev, V., 2022a. Deriving modernity signatures for PHP systems with static analysis. In: Ceccato, M., Roy, B., Ghafari, M. (Eds.), Proceedings of the 22nd IEEE International Working Conference on Source Code Analysis and Manipulation. IEEE, SCAM NIER, pp. 181–185. http://dx.doi.org/10.1109/SCAM55253.2022.00027.

van den Brink, W., et al., 2022b. PHP modernity signature. URL https://github.com/grammarware/modernity-php.

van Kemenade, H., Si, R., Dollenstein, Z., 2023. `Hugovk/top-pypi-packages`: Release 2023.01. http://dx.doi.org/10.5281/zenodo.7497599, Zenodo.

van Rossum, G., Drake Jr., F.L., 1995. Python Tutorial, vol. 620, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands.

Vavrová, N., Zaytsev, V., 2017. Does Python smell like Java? Art, Sci. Eng. Program. 1, 11–1–11–29. http://dx.doi.org/10.22152/programming-journal.org/2017/1/11.

Verhoef, C., 2002. Quantitative IT portfolio management. Science of Computer Programming 45 (1), 1–96. http://dx.doi.org/10.1016/S0167-6423(02)00106-5.

Vinju, J., 2021. Python-air. https://github.com/cwi-swat/python-air.

Wiese, E.S., Yen, M., Chen, A., Santos, L.A., Fox, A., 2017. Teaching students to recognize and implement good coding style. In: Proceedings of the Fourth ACM Conference on Learning At Scale. L@S, ACM, pp. 41–50. http://dx.doi.org/10.1145/3051457.3051469.

Zaytsev, V., 2011. Language convergence infrastructure. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (Eds.), Post-Proceedings of the Third International Summer School on Generative and Transformational Techniques in Software Engineering. GTTSE, In: LNCS, vol. 6491, Springer, pp. 481–497. http://dx.doi.org/10.1007/978-3-642-18023-1_16.

Zaytsev, V., 2016. Cotransforming grammars with shared packed parse forests. ECEASST Spec. Issue Sel. Revised Pap. Graph Comput. Models 73, http://dx.doi.org/10.14279/tuj.eceasst.73.1032.

Zaytsev, V., 2020. Modelling of language syntax and semantics: The case of the assembler compiler. In: Proceedings of the 16th European Conference on Modelling Foundations and Applications in the Journal of Object Technology, Vol. 19. ECMFA@JOT, http://dx.doi.org/10.5381/jot.2020.19.2.a5.

Zaytsev, V., Fabry, J., 2019. Fourth generation languages are technical debt. In: International Conference on Technical Debt. TechDebt, Extended Abstract, http://grammarware.net/text/2019/4gl-techdebt.pdf.

Zhang, Z., Xing, Z., Xia, X., Xu, X., Zhu, L., 2022. Making Python code idiomatic by automatic refactoring non-idiomatic python code with pythonic idioms. In: Roychoudhury, A., Cadar, C., Kim, M. (Eds.), Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE, ACM, pp. 696–708. http://dx.doi.org/10.1145/3540250.3549143.

Zhang, Z., Xing, Z., Xu, X., Zhu, L., 2023. Ridiom: Automatically refactoring non-idiomatic Python code with pythonic idioms. In: Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE), Companion Volume. IEEE, pp. 102–106. http://dx.doi.org/10.1109/ICSE-COMPANION58688.2023.00034.

Zolfani, S.H., Yazdani, M., Pamucar, D., Zaraté, P., 2020. A VIKOR and TOPSIS focused reanalysis of the MADM methods based on logarithmic normalization. Facta Univ., Ser. Mech. Eng. (FUME) 18 (3), http://dx.doi.org/10.22190/FUME191129016Z.

Zubcu, C., 2023. Effect of Normalization Techniques on Modernity Signatures in Source Code Analysis. Universiteit Twente, http://purl.utwente.nl/essays/96034.

Chris Admiraal is a Master student of the Computer Science programme at the University of Twente.

Wouter van den Brink a Master student of the Education and Scientific Communication programme at the University of Twente.

Marcus Gerhold is an Assistant Professor at the University of Twente, focusing his research on models of stochastic behaviour with domains ranging from predictive maintenance of safety-critical applications to procedural content generation in games.

Vadim Zaytsev is an Associate Professor and a Programme Director at the University of Twente, as well as the Editor-in-Chief of the SLEBoK initiative, with a research focus on grammars, software modelling, software evolution and legacy systems.

Cristian Zubcu is a 2023 graduate of a Bachelor programme in Technical Computer Science at the University of Twente.