Effect of Normalization Techniques on Modernity Signatures in Source Code Analysis

CRISTIAN ZUBCU, University of Twente, The Netherlands

Modernity signatures represent a novel way of assessing the modernity of a codebase. By quantifying the usage of language specific features, these signatures provide a measure of the degree to which the latest capabilities of a programming language are utilized. Normalization plays a crucial role in shaping the interpretation of these modernity signatures, highlighting various aspects of code evolution. However, the choice of normalization techniques and their implications have been somewhat overlooked in prior research. To bridge this gap, we present a study that scrutinizes the influence of various normalization methods, including Max, Max-Min, Vector, Z-score, and Log normalization on modernity signatures in Python. Through a thorough analysis, we reveal how each technique uniquely modifies the modernity signature, offering diverse insights into codebase evolution. These insights encompass aspects such as dominant language versions, feature distribution, and their shifts overtime. Our findings aim to assist developers in critically assessing their code's modernity and understanding the nuanced evolution of their codebase over time.

Additional Key Words and Phrases: Normalization, Modernity Signatures, Python, Code Evolution, Code Modernity, Source Code Analysis

1 INTRODUCTION

The concept of "modernity" has long been a subject of extensive discussion across a multitude of disciplines, including philosophy, sociology, and technology. In the context of software engineering, modernity represents the extent to which a software system's source code leverages the contemporary features and capabilities inherent in the respective programming language. Analyzing the degree of modernity in source code is critical for evaluating and comparing various aspects, such as quality, maintainability, and adherence to best practices across different programming languages [9].

Recent research conducted by Chris Admiraal and Wouter van den Brink, two students from the University of Twente, has significantly contributed to this domain by devising methods to calculate modernity signatures for the PHP and Python programming languages [1, 20]. In the context of their work, modernity signatures represent vectors of values, where each value corresponds to a language version and represents the number of features originating from that specific version.

Both van den Brink and Admiraal employ the use of normalization in transforming these raw modernity signature values into a format that facilitates effective comparison [18]. However, it is intriguing to note that the two different prior works employ different normalization techniques. Despite the significance of normalization in the context of computing the modernity signatures, the impact of the chosen normalization techniques on the results is not specifically investigated.

In light of this, the focus of our research is to assess how different normalization techniques affect these modernity signatures. Our findings could be pivotal, as by broadening the scope of normalization 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 them to make informed decisions regarding project development, and in the long run, lead to an improved technical quality of their code [13].

2 PAPER OUTLINE

In order to aid the reader in studying our research, we present the following outline: Section 3 discusses the previous studies dealing with modernity signatures and normalization methods. Our research and sub-research questions are laid out in Section 4. The experiment's method is explained in Section 5, which aims to answer these questions. Section 6 delves into the experiment setup, shares the results, and discusses their implications. Section 7 talks about possible issues that might affect our findings' validity. Finally, Section 8 presents the study's conclusion and outlines future research directions.

3 RELATED WORK

As mentioned in Chapter 1, this paper seeks to build upon the previously laid work of Van den Brink and Admiraal [1, 20]. Van den Brink, in his study, leveraged grammar usage statistics to generate the modernity signatures. He defined the grammar for attributed statements in PHP and analyzed its usage throughout a codebase. By examining the frequency of each type of attributed statement (function, class, trait, interface, enumeration), he attached annotations to indicate their relative usage. It's worth noting that van den Brink's work, co-authored by Gerhold and Zaytsev extended beyond the scope of University of Twente, reaching the 2022 SCAM conference [21].

Admiraal on the other hand, developed a tool, Pyternity, which utilizes another tool called Vermin [11] to identify and count the features from each Python version used. Both van den Brink and Admiraal's implementations allow for the data to be arranged in three-dimensional plots. This intuitive representation of feature distribution across time and language versions allow us to visually inspect different modernity signatures based on their plots, therefore facilitating the assessment of the varied effects of normalization on the signatures.

The field of software evolution has also been previously explored. For instance, Mens et al. have previously identified various challenges in software evolution [15]. These include the need for scalable tools to handle aging code and the requirement for a wide range of

TScIT 39, July 7, 2023, Enschede, The Netherlands
© 2023 University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

data to support claims about a code's evolution. Some research has even tied external factors like bug reports, version repositories, and documentation to the evolution of a codebase. An example of this is Girba et al.'s work, where they detail how developers themselves influence the evolution of their projects [8].

When addressing the topic of normalization, it is noteworthy that existing research has compared various techniques across a range of domains. For instance, Chakraborty et al. evaluated the effect of normalization in the context of Multi-Attribute Decision Making (MADM) problems [5]. In another study, Dubois et al. highlighted the significance of normalization in scaling protein data for medical research [6]. These studies underscore the pervasive importance of normalization techniques in different research contexts. However, to the best of our knowledge, a research gap exists when it comes to the application of normalization specifically to modernity signatures.

4 RESEARCH QUESTIONS

This study aims to assess the influence of different normalization techniques on modernity signatures, and determine if they might reveal new perspectives on a project's level of modernity. Our exploration will be guided by the subsequent research questions:

RQ1 *How do different normalization techniques impact modernity signatures?*

To further elaborate on **RQ1**, the following sub-question is presented:

RSQ1 *What normalization techniques are applicable for modernity signatures?*

By answering **RQ1** and **RSQ1**, we can hopefully comprehend the effect of each normalization technique on modernity signatures. However, it is also essential to understand the value of each normalization technique in assessing the evolution of a project's codebase. This leads us to the second research question:

RQ2 *What insights can different normalization techniques provide into a project's code evolution?*

5 METHODOLOGY

In this chapter, we outline the research methodology employed to address the research questions presented in Chapter 4. The methodology was designed to systematically investigate the effects of different normalization techniques on the interpretation of modernity signatures.

5.1 Implementation of the Algorithms

Following the review of prior research, it was essential to validate the reproducibility and robustness of the algorithms used by Van den Brink and Admiraal [3]. This was done by running the algorithms locally, thereby generating a new set of modernity plots, and comparing them with the plots of the primary study. Both van den Brink's and Admiraal's code can be found on GitHub [2, 19] alongside with the respective documentation.

5.2 Identification of Applicable Normalization Techniques

Section 1 underscored that the foundational studies implemented two distinct normalization techniques on the modernity signatures. Van den Brink, in his work, employed a method in which the signature was scaled by its maximum element, known as Max normalization. In contrast, Admiraal chose to normalize by dividing the signatures by the sum of their elements, a method known as Sum normalization. Both of these normalization techniques belong to the linear category, as delineated in the study by Camarinha-Matos et al [4].

While the prior studies adhered to linear methods, our investigation aims to expand the scope of analysis to the semi and non-linear methods, referenced in the work of Camarinha-Matos et al., as well as those identified through a comprehensive review of the existing literature in the field.

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 normalization techniques.

Our goal in this research is to ensure a multifaceted perspective to mitigate potential bias, thereby enhancing the applicability and robustness of our findings [17].

5.3 Implementation of Different Normalization Techniques

The next phase was to incorporate these identified normalization techniques into the existing codebases of van den Brink and Admiraal [2, 19]. This implies adjusting their current implementation for computing modernity signatures to include the additional normalization methods. Conveniently, both projects already contain plot generation code for visualizing modernity signatures over time in their respective GitHub repositories. This readily available code was adapted to incorporate the varying normalization methods under review.

5.4 Analysis of Different Normalization Techniques

Having the necessary code, the critical stage of this investigation involved scrutinizing and contrasting the plots generated as a result of each normalization 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, only one was included in the sample. This was done in order to increase the likelihood that the findings would generalize to new data [10].

For each normalization 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.

It is important to note that the analysis will be conducted through visual examination, which will involve a thorough assessment of the presence, magnitude and proportions of the peaks in each plot.

Peak presence involves identifying any novel peaks that have emerged in the normalized plot in comparison to the original or discerning any significant peaks that have vanished.

Next, peak magnitude will be examined. This process involves comparing the peaks' sizes. The objective here is to identify any instances where the peaks have grown larger or become smaller,

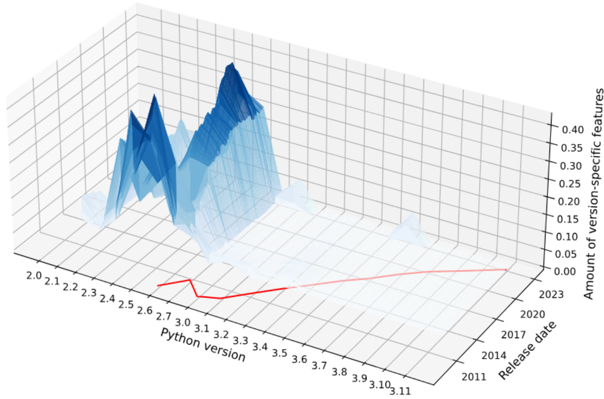

Fig. 1. *botocore* plot newly generated.

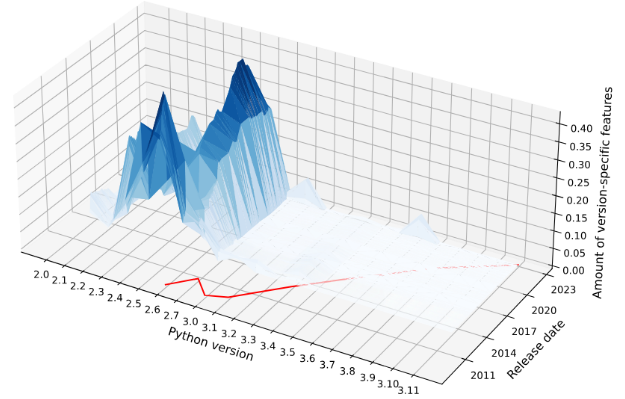

Fig. 2. *botocore* plot generated by Admiraal

marking significant differences in the plotted data's feature usage or distribution.

Finally, the examination of peak proportions involves comparing how the peaks in the normalized 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.

The color gradient of the plots was also considered for the comparison process. To illustrate, a plot with higher values would have a darker blue color and one with lower values would have predominantly lighter blue shades. However, the color of the graphs depends on the magnitude of the peaks, which was already addressed above. Therefore, we will not explicitly include the color gradient of the graphs in our methodology.

In short, analyzing the peak presence, magnitude, and proportions should give us a well-rounded view of the data distribution within the modernity signatures.

6 EXPERIMENT

The following section describes the steps followed in the experimental part of our research. Section 6.1 describes the setup of the experiment, including the selected dataset and normalization techniques to be compared. Section 6.2 follows, showing the outcomes of our comparisons. In Section 6.3, we talk about how different normalization techniques affect the results. Finally, in Section 6.4, we discuss what our findings mean for the field of software evolution.

6.1 Setup

As referenced in Section 5.1, the source code for both the PHP and Python projects is publicly available on GitHub [2, 19]. The repositories were cloned on a local computational setup, running Windows 11 (22H2), equipped with an Intel Core i7-10750H processor, operating at a base frequency of 2.6 GHz. In conjunction with the respective research papers [1, 20], the source code is generally intelligible. However, difficulties were encountered in executing the PHP project due to the incomplete source code files from the GitHub repository. Consequently, the reproducibility of van Den Brink's results was found to be compromised. This unfortunate development prompted a shift in the research focus, concentrating solely on Admiraal's work for the remainder of this study. As such, ten PyPI projects from Admiraal's initial dataset were integrated into the research data. Specifically, the dataset for this study comprises the following PyPI projects: *attrs*, *boto3*, *botocore*, *charset-normalizer*, *fsspec*, *google-api-core*, *Jinja2*, *requests*, *urllib3*, and *wheel*.

We re-calculated the signatures of the ten projects in our dataset using Admiraal's original code and documentation. All newly generated plots look mostly the same as their original counterparts. The main difference is that the newly generated plots display more recent modernity signatures. This is due to the fact that the original study was performed three months prior to the current study. In that interval, new versions of the projects were released, which the algorithm subsequently picked up and included. An example of this can be seen in the plot for the *botocore* project. The newly generated plot is shown in Figure 1. As a reference, Figure 2 shows the plot for the same project, generated at the time of Admiraal's research.

The work of Admiraal includes a feature that allows the setting of a maximum release date. By choosing a date closer to the time of Admiraal's study, it was possible to generate plots that match the original ones. This outcome strongly affirms the reproducibility of Admiraal's code and lays a solid groundwork for the subsequent stages of our investigation.

Setting up the experiment also consisted in identifying the normalization techniques which might be applicable in our research. To tackle that, we've strategically selected normalization methods from three distinct categories as identified by Camarinha-Matos et al [4]. These categories include linear, semi-linear and non-linear methods.

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 [16].

We also employed the Vector normalization method from the semi-linear category, which accounts for the direction and magnitude of data vectors, offering a different perspective on the data [4].

Table 1. Normalization Formulas

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	$x_i' = \frac{\ln x_i}{\ln \prod_{m}^{i=1} x_i}$

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 [22].

Finally, we incorporated the Z-score semi-linear normalization method, as highlighted in the research of Singh et al [16]. 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 normalization method in our repertoire. It's important to acknowledge that within each category, Camarinha-Matos et al. describe additional normalization techniques which could potentially apply in our study. However, for the purpose of this research, they were excluded in order to minimize the computational time and potential complexity in implementation and interpretation. In a similar vein, Singh et al. have elucidated on the existence of other categories of normalization 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 [16]. Therefore, the decimal, sigmoidal and tanh-based methods are excluded.

Given that all the required formulas for the selected normalization methods are known and listed in Table 1, it was straightforward to implement them in code. In Admiraal's GitHub repository [2] a *main.py* file can be found, which contains the code directly responsible for normalizing the modernity signatures.

The implementation of the Max and Max-Min normalization 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 utilized. Specifically, the *np.mean()* and *np.std()* functions are employed to calculate the mean and standard deviation. For the Log normalization technique, the math library's *math.log1p()* function was used to calculate the logarithms.

6.2 Results

6.2.1 Max normalization. Upon applying Max normalization, a distinct increase in peak magnitude was observed across all ten projects within the Max normalized 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 normalized project plots.

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

6.2.3 Vector normalization. Post Vector normalization, 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 normalized project plots.

6.2.4 Z-score normalization. The results derived from the Z-score normalization paralleled those from Vector normalization. The peak magnitudes and proportions in the Z-score normalized plots mirrored the changes observed in the Vector normalized plots.

6.2.5 Log normalization. Log normalization 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 normalization was that new peaks appeared in every plot.

An overview of the normalization methods and the number of projects in which the peak increase, altered proportions, and emergence of new peaks were observed, is presented in Table 2.

Table 2. Normalization effects on the plots

Method	Peak Inc.	Prop. Alt.	New Peaks
Max/Max-Min	10/10	8/10	0/10
Vector/Z-score	8/10	2/10	0/10
Log	10/10	10/10	10/10

6.3 Discussion

6.3.1 Max normalization. The effect of Max normalization can be explained by the nature of the technique itself, which scales each value of the signature by its maximum element, resulting in a normalized signature with values ranging between zero and one [16]. One example of this can be seen in the Max-normalized plot for the *botocore* project, as illustrated in Figure 3. Comparing this with the original plot in Figure 1, it becomes evident that the values for the normalized 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-normalized plot for the *wheel* project, as shown in Figure 5. 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 Figure 6, the values for Python version 2.1 begin at a peak and then gradually decrease. Conversely, in the Max-normalized plot, the initial peak for Python version 2.1 remains at a maximum, only

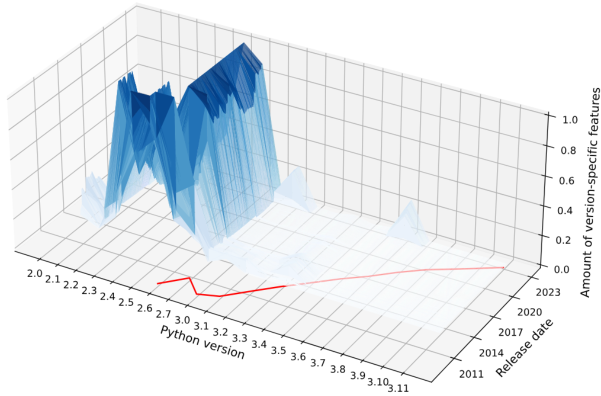

Fig. 3. *botocore* Max plot

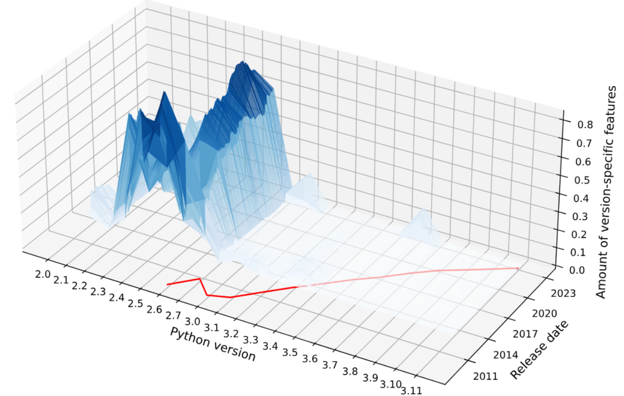

Fig. 4. *botocore* Vector plot

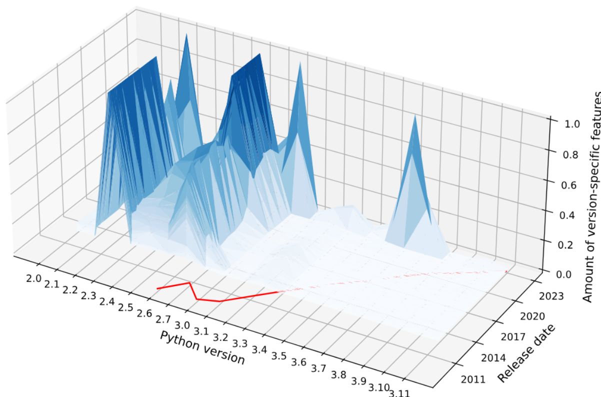

Fig. 5. *wheel* Max plot

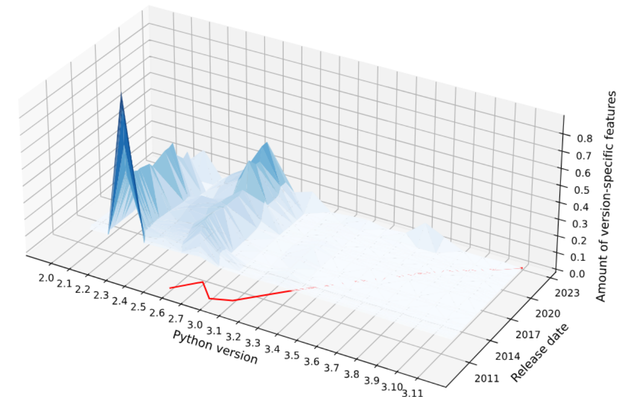

Fig. 6. *wheel* Original plot

decreasing when the feature count of Python 2.5 matches it, thereby leading to a noticeable shift in the peak values.

On closer examination, it was observed that plots with lower original values underwent a more drastic transformation. This is primarily due to the character of Max normalization, 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 normalization effectively accentuates the Python 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 normalized scale, it allows a clearer and more immediate identification of which Python version is predominant at each point in time.

6.3.2 Max-Min normalization. Max-Min normalization, akin to Max normalization, produced similar transformations on the plots within our dataset. Both these techniques are underpinned by the same principle: linear rescaling of data to fit a designated range [16]. 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 normalization, the minimum value in the data range becomes zero, essentially aligning the results with those obtained through Max normalization. Therefore, despite the slightly different methodology, the visual outcomes produced by Max and Max-Min normalizations ended up being virtually indistinguishable.

6.3.3 Vector normalization. The influence of Vector normalization on the projects within our dataset demonstrated varied outcomes. In total, eight out of the ten projects exhibited an amplified peak magnitude to some extent. 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 normalization were *wheel* and *requests*. The *wheel* project plot (see Figure 6), which had earlier exhibited significant changes under Max normalization, also showed a considerable difference in its Vector normalized plot (see Figure 9). Although the peak values corresponding to Python versions 2.1 and 2.5 significantly increased in the Vector normalized plot compared to the original, they did not stay at a constant maximum as was the case in the Max normalized plot. Instead, the peaks displayed a more gradual increase and decrease.

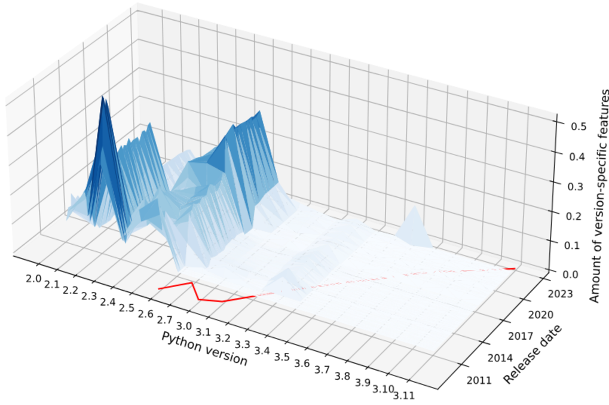

Fig. 7. *requests* Original plot

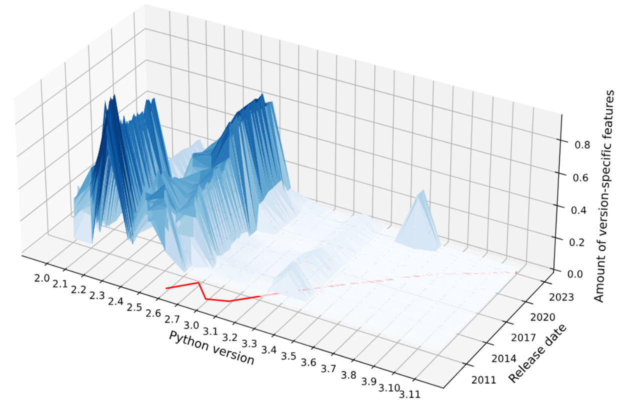

Fig. 8. *requests* Vector plot

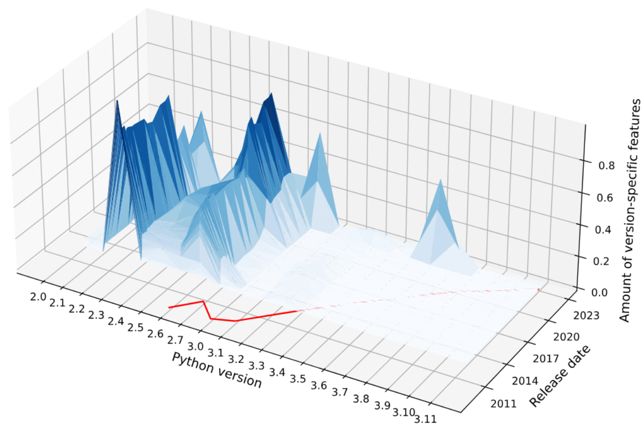

Fig. 9. *wheel* Vector plot

Upon looking for reasons why only *wheel* and *requests* were the only projects considerably affected by Vector normalization, it was discovered that both of these projects displayed similar characteristics in their original plots, as shown in Figure 6 and 7. As previously mentioned in the Section 6.3.2, 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 [12]. The effect of Vector normalization, however, was not uniformly witnessed across all projects within our dataset. A prime example of this is the botocore project, where the Vector normalized plot (refer to Figure 4) is strikingly similar to its original plot (refer to Figure 1). 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 normalized plot, the influence of Python version 2.4 features escalates steadily, mirroring the pattern observed in the original plot. This could mean that Vector normalization accentuates shifts in feature contributions from older to newer Python versions over time.

These observations imply that Vector normalization serves to emphasize 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 normalization may be less discernible. In contrast, for projects that display a shifting balance of feature contributions from different Python versions over time, Vector normalization can significantly amplify these changes.

6.3.4 Z-score normalization. Z-score normalization produced outcomes identical to those induced by Vector normalization. Both these techniques, despite having different mathematical formulas, effectively rescale the original feature vectors to have standardized properties. Z-score normalization transforms the data to have a mean of zero and a standard deviation of one [16]. On the other hand, Vector normalization scales the vectors to have a unit length [5]. 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 normalization methods provide an exactly similar view of the feature distribution across different Python versions.

6.3.5 Log normalization. The application of Log normalization to the project plots in our dataset resulted in an uniform effect on all project plots.

This effect is evident in the *urllib3* project, which was taken as an example. In the original plot of this project (see Figure 10), 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

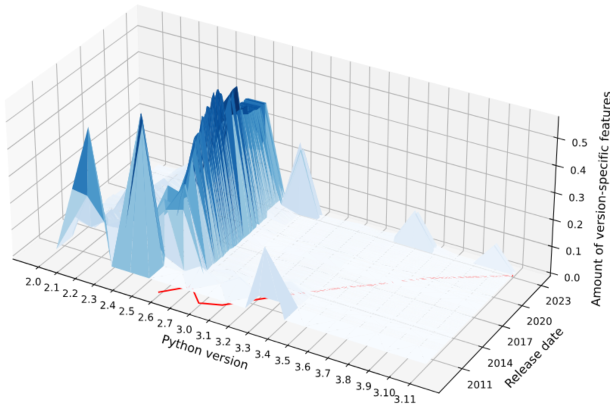

Fig. 10. *urllib3* Original plot

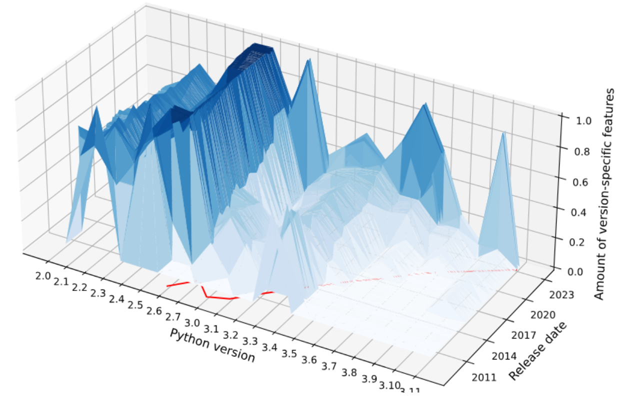

Fig. 11. *urllib3* Log plot

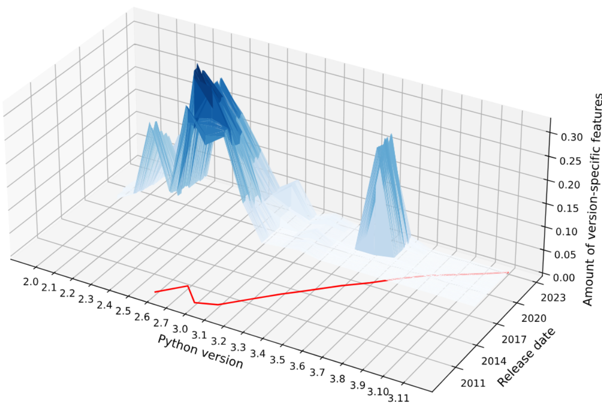

Fig. 12. *google-api-core* Original plot

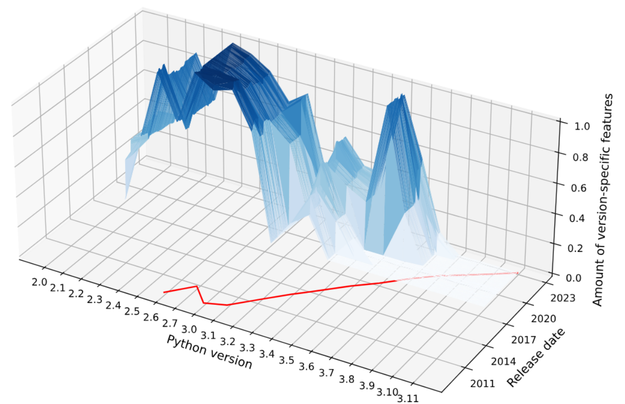

Fig. 13. *google-api-core* Log plot

changes significantly when subjected to Log normalization (see Figure 11).

The slight peaks corresponding to Python versions 3.1, 3.7, and 3.10 become considerably more pronounced in the Log normalized plot, reflecting substantially higher values. In addition, Log normalization 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 as well. In the original plot (refer to Figure 12), the visualization prominently features three peaks linked to Python versions 2.1, 2.3, and 3.5. However, upon the application of Log normalization (as depicted in Figure 13), 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 normalization 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.

6.4 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 [15]. Section 6.3 discusses how different normalization 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.

This goes to show that there is not a single best normalization 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's worth stating that the process of normalizing the signatures using different techniques to gain new insights isn't exclusive to Python. In theory, as long as there is a method of obtaining a modernity signature for a given programming language, the normalization process is applicable with any of the techniques discussed in this paper. After all, modernity signatures are just sets of numbers, regardless of the programming language. This will be addressed further in Section 8.

The data produced through our method can also be transformed into meaningful metrics, thereby offering deeper insights into the trajectory of code evolution [14]. For instance, using the Max and Max-Min methods the stability of a codebase can be inferred. That is, 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 [15].

7 THREATS TO VALIDITY

7.1 Internal 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 normalization 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 normalization method brings. Machine learning could also be employed to automatically identify patterns in the normalized data. Due to lack of remaining time to spend on this research, the mentioned ideas were not implemented, however, they present an interesting take on the internal validity threat.

7.2 External validity

The selection of the projects to be studied is another possible threat to the external validity of our work. Section 5.4 explains that the projects were selected such that none of them resemble each other, in order to reduce selection bias. However, this doesn't mean that our selection of projects adequately represent all the types and shapes of modernity signature plots. This means that 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 are making visual comparisons, adding more projects to our analysis would notably increase the time complexity of our research.

8 CONCLUSION AND FUTURE WORK

In this section, we answer the research and sub-research questions introduced in Chapter 4.

In response to **RQ1**, our investigation reveals that different normalization techniques distinctly alter the modernity signatures. Max and Max-Min methods predominantly emphasize the prevailing Python version in each period, while the Vector and Z-score techniques underscore dynamic transitions in feature usage over time. Log normalization highlights minor variations in feature usage, accentuating even the less prominent versions.

RSQ1 was addressed in Section 6.1. We determined that five specific normalization techniques - Max, Max-Min, Vector, Z-score, and Log normalization - 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 complex parameter setup.

RQ2 was addressed in Section 6.4. This study reveals that different normalization 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.

For future research, we propose extending our study to include a wider array of programming languages, which would enable a better understanding of the consistency of normalization technique impacts across different languages. Java for example, presents an interesting case, as Dyer et al. specify in their work that most of it's features see limited use in the current industry [7]. Therefore, it would be interesting to study the changes over time in Java projects using the different normalization methods from this study. This could also help determine if these new findings match up with the results of our current research.

REFERENCES

[1] C.P. Admiraal. 2023. Calculating the modernity of popular python projects. http://essay.utwente.nl/94375/

[2] C.P. Admiraal. 2023. Pyternity. https://github.com/grammarware/modernity-python

[3] Fabien C. Y. Benureau and Nicolas P. Rougier. 2018. Re-run, Repeat, Reproduce, Reuse, Replicate: Transforming Code into Scientific Contributions. *Frontiers in Neuroinformatics* 11 (2018). https://doi.org/10.3389/fninf.2017.00069

[4] Luis M. Camarinha-Matos, Nastaran Farhadi, Fábio Lopes, and Helena Pereira (Eds.). 2020. *Technological Innovation for Life Improvement.* Springer International Publishing. https://doi.org/10.1007/978-3-030-45124-0

[5] Subrata Chakraborty and Chung-Hsing Yeh. 2009. A simulation comparison of normalization procedures for TOPSIS. In *2009 International Conference on Computers Industrial Engineering.* 1815–1820. https://doi.org/10.1109/ICCIE.2009.5223811

[6] Etienne Dubois, Antonio Núñez Galindo, Loïc Dayon, and Ornella Cominetti. 2020. Comparison of normalization methods in clinical research applications of mass spectrometry-based proteomics. In *2020 IEEE Conference on Computational Intelligence in Bioinformatics and Computational Biology (CIBCB).* 1–10. https://doi.org/10.1109/CIBCB48159.2020.9277702

[7] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. 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* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 779–790. https://doi.org/10.1145/2568225.2568295

[8] T. Girba, A. Kuhn, M. Seeberger, and S. Ducasse. 2005. How developers drive software evolution. In *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*. 113–122. https://doi.org/10.1109/IWPSE.2005.21

[9] grammarware. 2021. Codebase Modernity Meter. https://github.com/grammarware/project/blob/main/bsc/Codebase_Modernity_Meter.md

[10] David J. Hand. 2006. Classifier Technology and the Illusion of Progress. *Statist. Sci.* 21, 1 (2006), 1 – 14. https://doi.org/10.1214/088342306000000060

[11] Morten Kristensen. 2018. Vermin. https://pypi.org/project/vermin/

[12] Brian A. Malloy and James F. Power. 2018. An empirical analysis of the transition from Python 2 to Python 3. *Empirical Software Engineering* 24, 2 (July 2018), 751–778. https://doi.org/10.1007/s10664-018-9637-2

[13] Likoebe M Maruping, Xiaojun Zhang, and Viswanath Venkatesh. 2009. Role of collective ownership and coding standards in coordinating expertise in software project teams. *European Journal of Information Systems* 18, 4 (Aug. 2009), 355–371. https://doi.org/10.1057/ejis.2009.24

[14] Tom Mens and Serge Demeyer. 2001. Future Trends in Software Evolution Metrics. *International Workshop on Principles of Software Evolution (IWPSE)*, 83–86. https://doi.org/10.1145/602461.602476

[15] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri. 2005. Challenges in software evolution. In *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*. 13–22. https://doi.org/10.1109/IWPSE.2005.7

[16] Dalwinder Singh and Birmohan Singh. 2020. Investigating the impact of data normalization on classification performance. *Applied Soft Computing* 97 (2020), 105524. https://doi.org/10.1016/j.asoc.2019.105524

[17] Joanna Smith and Helen Noble. 2014. Bias in research. *Evid. Based. Nurs.* 17, 4 (Oct. 2014), 100–101.

[18] Nazanin Vafaei, Rita A. Ribeiro, and Luis M. Camarinha-Matos. 2022. Assessing Normalization Techniques for Simple Additive Weighting Method. *Procedia Computer Science* 199 (2022), 1229–1236. https://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.

[19] W. van den Brink. 2022. PHP Modernity Signature. https://github.com/grammarware/modernity-php

[20] W. van den Brink. 2022. Weighed and found legacy : modernity signatures for PHP systems using static analysis. http://essay.utwente.nl/91794/

[21] Wouter Van den Brink, Marcus Gerhold, and Vadim Zaytsev. 2022. Deriving Modernity Signatures for PHP Systems with Static Analysis. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 181–185. https://doi.org/10.1109/SCAM55253.2022.00027

[22] Sarfaraz Hashemkhani Zolfani, Morteza Yazdani, Dragan Pamucar, and Pascale Zaraté. 2020. A VIKOR and TOPSIS focused reanalysis of the MADM methods based on logarithmic normalization. *CoRR* abs/2006.08150 (2020). arXiv:2006.08150 https://arxiv.org/abs/2006.08150