Leveraging Deep Learning for Python Version Identification

Marcus Gerhold[1], Lola Solovyeva[2] and Vadim Zaytsev[1,2]

[1]*Formal Methods & Tools, University of Twente, Enschede, the Netherlands*
[2]*Computer Science, University of Twente, Enschede, the Netherlands*

Abstract
Python, recognised for its dynamic and adaptable nature, has found widespread application in a myriad of projects. As the language evolves, determining the Python version employed in a project becomes pivotal to ensure compatibility and facilitate maintenance. Deep learning (DL) has emerged as a promising tool to automate this process. In this research, we assess various DL techniques in determining the minimum Python version required for a given project. We explore the complexities of handling Python data and the quest for optimal text segmentation techniques to achieve high classification accuracy, particularly for lengthy files. Our experimental results show that, although DL algorithms exhibit a low misclassification rate with short code snippets, their performance significantly falters with long files. This research provides valuable insights into the challenges associated with utilising programming languages for deep learning models and suggests potential solutions for addressing these issues.

Keywords
Deep Learning, CodeBERT, Python, version identification

1. Introduction

Python continues to hold its position as one of the most widely used programming languages of our generation. As indicated by JetBrains, Stack Overflow, and IEEE Spectrum, Python consistently ranks among the top three programming languages preferred by developers and claims the top spot when it comes to researchers' preferences. Python has undergone a series of significant evolutions and version updates since its inception [1]. The previous findings have shown that during the breakthrough of Python 3 developers had not fully embraced the transition to a newer version. Instead, they opted to maintain compatibility with both Python 2 and 3, limiting themselves to a subset of the language governed by the decreasing set of shared features between Python 2 and 3 [1]. This closes the door for compatibility with other projects that fully transitioned to newer versions since Python does not maintain backward compatibility [2]. Exploiting projects with an older version can lead to software quality issues such as increased complexity of the code, security vulnerabilities, and performance limitations. While certain Python projects indicate the necessary version for their execution, this requirement may not always represent the actual minimum version. In practice, developers do not always utilise the functionalities of the version they use. There is a very limited number

BENEVOL'23: The 22nd Belgium-Netherlands Software Evolution Workshop; Nijmegen, 27–28 November 2023
m.gerhold@utwente.nl (M. Gerhold); o.solovyeva@student.utwente.nl (L. Solovyeva); vadim@grammarware.net (V. Zaytsev)

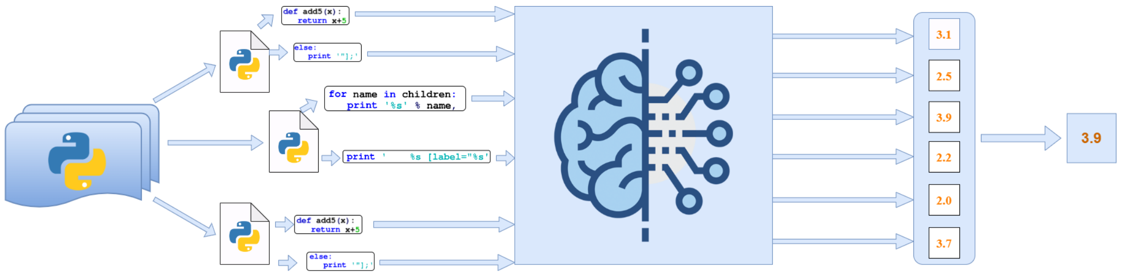


Figure 1: Research pipeline for finding minimal version for a Python project using deep neural network.

of existing solutions for how to determine a minimum required version for a Python code. The prevailing method is often a trial-and-error approach, where one relies on their prior experience and familiarity with Python features to gauge the necessary version. Other existing solutions involve parsing the code and then cross-referencing it with internal dictionaries.

At the same time, the increased utilisation of deep learning techniques has become more prominent in the realm of software engineering and development. It has proven highly beneficial in tasks such as identifying code smells [3], code summarisation [4], and detecting code clones [5]. In this research, we investigate the ability of deep learning techniques to find subtle differences between various versions of Python language. To find a minimal required Python version for the codebase, we propose to train a deep learning model that distinguishes between Python minor versions, amounting to a current count of 20 distinct classes. Figure 1 demonstrates the pipeline of our proposed approach. The first step requires splitting a Python project into individual files, which are further divided into chunks of text. Each chunk is then fed into the classifier, yielding the minimal version required for a successful compilation. The results for each chunk are saved into a list. The maximum version in that list represent the minimal required version for the chosen Python project. Our approach answers the following two research questions:

1. Which DL model provides the highest level of accuracy when classifying Python versions?
2. What text segmentation approach will effectively capture the characteristics of the file for its accurate classification?

2. Related work

There has been limited attention and research dedicated to the problem of identifying required Python versions for the file or a project. An existing tool, known as Vermin[1], has the capability to determine the minimum required Python version. Vermin accomplishes this by parsing code into an abstract syntax tree and subsequently traversing it while comparing against internal dictionaries with 3676 rules. Nevertheless, it may still produce erroneous results and is not scalable for major projects [6]. Additionally, there is a Chrome extension named PyVerDetector, which empowers users to select a specific Python version and validate the compatibility of code snippets on Stack Overflow [2]. It generates error messages for any inconsistencies found,

[1]https://github.com/netromdk/vermin#vermin

parsing the code snippets and highlighting versioning issues, while also suggesting a list of Python versions that can execute each code snippet. Nonetheless, PyVerDetector is limited to recognising major Python versions and does not possess the capability to differentiate between minor version variations. Another tool that was developed with the same limitation is PyComply, which is a Python compliance analyser [1]. It was developed to assess and quantify the extent to which Python 3 features are utilised, including their adoption rate and the context in which they are applied. At the heart of PyComply lies the foundation of its grammar formalism, which serves to define the Python syntax. Additionally, parser actions have been seamlessly incorporated into this grammar to aid in recognising the distinctive features of Python 3.
Previously deep learning techniques were applied to Python data for various reasons. Akimova et al. [7] created a dataset PyTraceBugs that serves the purpose of training, validating, and assessing large-scale deep learning models with the specific objective of identifying a distinct category of low-level bugs present in source code snippets. Furthermore, Alhefdhi et al. [8] applied Neural Machine Translation to Python data for pseudo-code generation. Nonetheless, there is no dataset that has pairs of Python code with their corresponding versions.

3. Corpus construction and pre-processing

There is no existing corpus that contains pairs of Python code snippets and their corresponding version. Thus, there is a need to create a dataset, that would consist of code examples for each of the Python versions. We use Vermin for labeling the snippets since the version provided on PyPI[2] is set for the entire project. So, for some files of the project, the version listed on PyPI is not necessarily a minimal one.
We collected Python code samples by downloading 50 popular Python projects from PyPI, considering each project's multiple releases. We focused on Python files, excluding those in other languages, and removed comments. Using a dedicated Python package, we generated Abstract Syntax Trees (ASTs) for each file, discarding unparsable ones. The Vermin tool helped us determine the minimal version required for the successful compilation of individual AST nodes, aiming to find distinctive version features. Terminal nodes, typically representing variables or numeric values, were excluded as they lack substantial version-differentiating information. So, the resulting dataset is the mapping between code snippets, which represent distinctive features according to Vermin, and its corresponding version.

Table 1 presents the number of instances for each class. The dataset is imbalanced since some Python versions are more commonly used than others. This can drastically impact a training process and classification results. To deal with this issue, we apply a widely-used approach to synthesising new class instances called Synthetic Minority Oversampling TEchnique (SMOTE). Since some of the classes have a minuscule number of instances, oversampling is more beneficial.

[2]https://pypi.org/

Version	2.0	2.1	2.2	2.3	2.4	2.5	2.6	2.7	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7	3.8	3.9	3.10	3.11
Number of Instances	1200199	192	13985	4719	16831	14151	26685	7166	20741	74	514	2317	425	6538	25146	184	722	63	1792	36

Table 1
Number of instances per class of minor Python version.

	LSTM	TCN	TextCNN	BERT	CodeBERT	XLNet
Word2Vec	🟩	🟩	🟩	🟥	🟥	🟥
CodeBERT	🟩	🟩	🟩	🟩	🟩	🟥
XLNet	🟥	🟥	🟥	🟥	🟥	🟩

Table 2
Combination of word embeddings and classifiers. Green indicates the combination was used, whereas red indicates that it was not.

4. Performance evaluation

As a final step of the experiment, our objective is to assess the performance of the models employed in this study. Our primary goal is to identify a model, that is capable of effectively categorising a code snippet with its associated minimum required version. This step holds paramount importance, as it is crucial for a model to exhibit a high classification accuracy on a per-code snippet basis. This significance arises from the fact that a single file can be divided into multiple code snippets, meaning that an incorrect classification of just one instance could result in the misclassification of the entire file.
We use some of the most common metrics to evaluate the performance of text classifiers: accuracy, recall, precision, F1-score, and confusion matrix. We also employed balanced accuracy, which is a variation of the standard accuracy but it takes into account the class distribution in the dataset. For a multiclass problem, it is an average of recalls per class.
Besides evaluating the model on the test set, which consists of short snippets, we also conduct evaluations on lengthy files that are entirely distinct from both the training and testing data, forming an entirely separate dataset, that underwent the same collection approach as a dataset for short code snippets. We explore various segmentation techniques to identify the most effective approach for achieving precise file classification based on the ground truth labels. We consider the following methods:

- **First *n* words** → feed only the first n words into the model, where n is a hyperparameter.
- **Per line** → split files into lines and feed each line into the model.
- **Per part** → split files into parts of length n, where n is a hyperparameter, and feed each line into the model.
- **Import statements** → feed each import statement of the file into the model.
- **AST nodes** → parse the file into the AST and feed the nodes into the model.

Model	Accuracy	Balanced Accuracy	Precision	Recall	F1-score
Word2Vec+LSTM	0.62	0.55	0.65	0.62	0.63
Word2Vec+TCN	0.51	0.42	0.55	0.55	0.55
Word2Vec+TextCNN	0.56	0.46	0.59	0.56	0.57
CodeBERT+LSTM	**0.93**	**0.92**	**0.93**	**0.93**	**0.93**
CodeBERT+TCN	0.90	0.89	0.91	0.90	0.90
CodeBERT+TextCNN	0.92	0.90	0.92	0.92	0.92
CodeBERT+BERT	0.92	0.90	0.92	0.91	0.91
CodeBERT	0.92	0.89	0.92	0.92	0.92
XLNet	0.92	0.89	0.92	0.92	0.92

Table 3
Results of each model for accuracy, balanced accuracy, precision, recall, and F1-score on the test set of short code snippets.

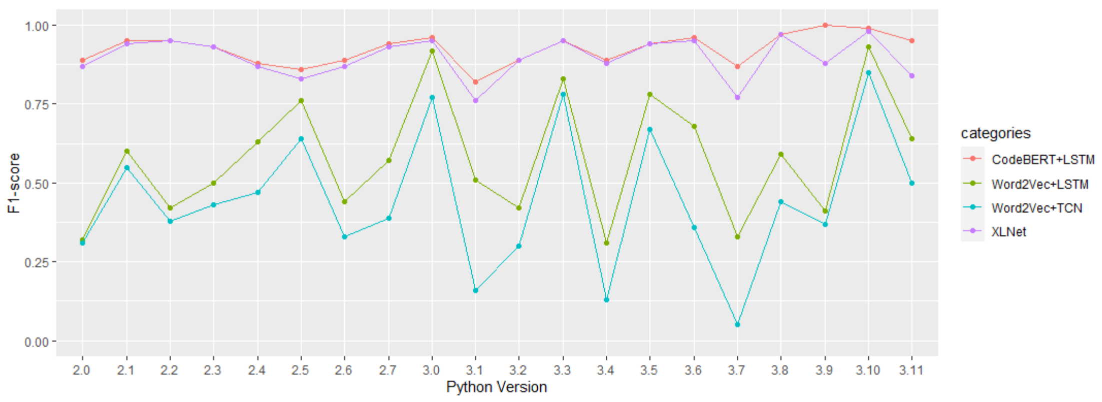


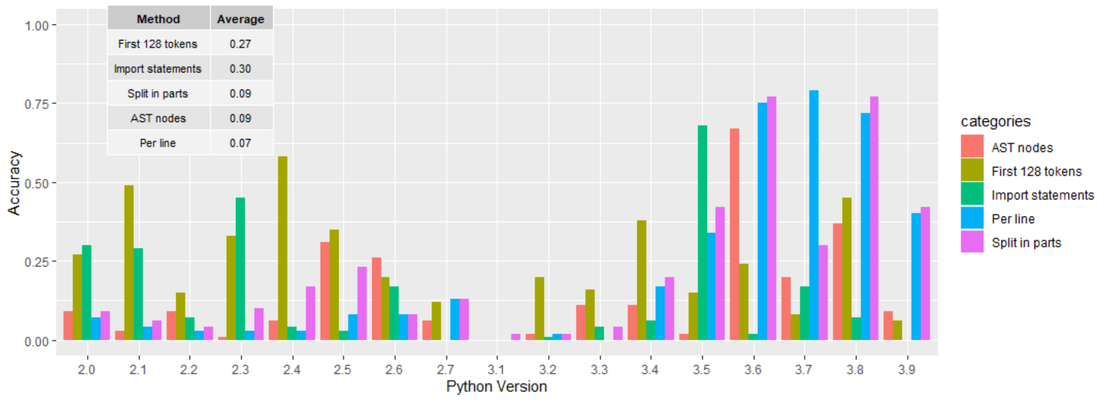
Figure 2: F1 score of four models per each Python version.

5. Discussion

As indicated in Table 2, nine models underwent training and evaluation on two datasets to showcase their ability to distinguish among 20 Python versions, with the aim of identifying the minimum version needed for a given project. In the following section, we will illuminate key findings derived from the experimental outcomes and elucidate the challenges and constraints associated with handling Python data.

5.1. Highlights on model behaviour

Table 3 presents the results from the evaluation, demonstrating the superiority of the LSTM model with CodeBERT embedding, achieving 93% for each metric. Figure 2 illustrates the F1-scores for individual classes attained by four models, two of which were top performers while the other two performed poorly. Choosing models in such a way demonstrates the contrast of their performance. The findings clearly indicate that replacing Word2Vec with CodeBERT embeddings leads to noticeable improvements in all metrics. This demonstrates that using domain-specific embeddings like CodeBERT greatly enhances the model's ability to classify instances accurately across all categories. CodeBERT's strength in understanding contextual nuances and capturing distant token relationships is key in structured text like source code [9]. Coupled with LSTM, this model excels in handling sequential data, enabling it to retain tokens in memory over extended periods, which is particularly beneficial for programming languages with dependencies throughout the code [10]. An interesting finding reveals lower validation accuracy for transformers compared to LSTM and TCN, consistent with previous research [11]. BERT, in particular, exhibits reduced performance on smaller datasets, due to its original training on extensive corpora. This limitation is emphasised by the scarcity of certain Python versions in PyPI projects, resulting in a limited number of instances for specific classes, rendering the dataset insufficient for precise transformer model training. Nevertheless, the transformer models still achieve high accuracy. This same observation has been reported previously, suggesting that it may be attributed to the inadequacy of the testing data [12]. Since the model was trained on samples with distinctive version features, the test set may contain instances that share structural and lexical similarities with the training data. This similarity arises from consistent function and library names across versions. While not identical, training and test instances resemble each other due to the limited source code vocabulary [13].



Method	Average
First 128 tokens	0.27
Import statements	0.30
Split in parts	0.09
AST nodes	0.09
Per line	0.07

Figure 3: The accuracy of prediction for each class by LSTM model with CodeBERT embedding using various text segmentation techniques listed in the categories.

5.2. Highlights on text segmentation

To compare the results of various text segmentation techniques, we chose the best-performing model, which in this case is LSTM with CodeBERT embedding, and used it to predict the minimal version of the file as proposed in Figure 1. The results of the evaluation can be found in Figure 3. We tested five techniques for segmenting the file, since BERT-like models have an input limit of 512 tokens, aiming to find the method that maximises the accurate classification.
Analysing import statements to predict the minimal Python version yields the highest accuracy among the methods used in this study. Import statements are intuitive indicators of version requirements, but the model's limited awareness of all libraries and custom-developed modules

contributes to an accuracy ceiling of 30%. This also holds true for imported modules that users may have personally developed. The model may struggle to recognise the version requirement for importing such modules, mainly because they are infrequently encountered in the training data, making it challenging for the model to grasp these nuances. Additionally, when rare modules or absent import statements are encountered, the model relies solely on syntactic features to distinguish versions. The second most successful method, truncating files to the first 128 words, achieved 27% accuracy. This lower accuracy is due to discarding potentially significant information found in the rest of the file, as we only analyse the beginning.
Using the model on file chunks yielded unfavorable results due to the challenge of balancing information and minimising misclassification. Increasing the number of chunks raises the risk of misclassifying at least one, which could lead to incorrect version assignments for the entire file. This explains the lower accuracy in the other three methods.

5.3. Highlights on difficulties

One of the many challenges is an infinite vocabulary span, signifying endless possibilities of potential names for identifiers [10, 13]. The corpus must be big enough to cover all the possibilities of the variable name. Nevertheless, even in such circumstances, the model might encounter an unfamiliar token, which can significantly undermine its overall performance.
Incorporating natural language within source code, whether in variable names, strings, print statements, or error messages, can significantly impact the model's performance. This aligns with a study on code summarisation, where the presence of natural language improved summarisation accuracy [9]. However, in our case, it introduces unwanted noise, negatively affecting performance. Our goal is to distinguish between Python versions by identifying unique characteristics, so it is crucial to isolate these features from any noise to ensure accurate classification.
Another Python-related challenge involves the potential use of function names introduced in newer versions as variable names in older versions, or even introducing a variable with the same name as a function. For instance, consider the `match`[3] function introduced in Python 3.10. In all versions prior to 3.10, it is possible to have any identifier with the name `match`. This scenario can create the misconception that the occurrence of `match` is equally probable across all versions, resulting in no informational gain for the model. This surely can be prevented if the model captures the structural difference between the introduction of the variable `match` and the use of pattern matching. As mentioned earlier, transformers such as BERT utilise attention mechanisms to comprehend token relationships. However, they have input capacity limits, requiring us to truncate or segment files. The challenge is that improper segmentation can disrupt unique feature structures, potentially leading to misclassification due to lost context.

6. Conclusion

We examined nine deep-learning models for Python version classification. LSTM with CodeBERT embedding yielded the highest accuracy when applied to a dataset containing Python features. Furthermore, import statements demonstrated to be the most effective technique to capture the information about a required version of the file. Nevertheless, classifying lengthy

[3]https://docs.python.org/3/whatsnew/3.10.html

files with deep learning remains a challenging task due to certain issues, including input limitations, overlap between new and old versions, the presence of natural language in code, and the wide variability in variable names. Future improvements include masking the natural language in the code, which will reduce the noise in the data, and identifying suitable alternatives for unseen variable names, so the model can make more accurate predictions based on the data it has seen. Additionally, expanding the training corpus will help to minimise the likelihood of encountering unseen modules and libraries.

References

[1] B. A. Malloy, J. F. Power, Quantifying the Transition from Python 2 to 3: An Empirical Study of Python Applications, in: Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2017, pp. 314–323. doi:10.1109/ESEM.2017.45.

[2] S. Yang, T. Kanda, D. Pizzolotto, D. M. German, Y. Higo, PyVerDetector: A Chrome Extension Detecting the Python Version of Stack Overflow Code Snippets, in: Proceedings of the 31st IEEE/ACM International Conference on Program Comprehension (ICPC), 2023, pp. 25–29. doi:10.1109/ICPC58990.2023.00013.

[3] S. Tarwani, A. Chug, Application of Deep Learning models for Code Smell Prediction, in: Proceedings of the 10th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO), 2022, pp. 1–5. doi:10.1109/ICRITO56286.2022.9965048.

[4] T. Zhu, Z. Li, M. Pan, C. Shi, T. Zhang, Y. Pei, X. Li, Revisiting Information Retrieval and Deep Learning Approaches for Code Summarization, in: Proceedings of the International Conference on Intelligent Computing and Human-Computer Interaction (ICHCI), 2023, pp. 328–329. doi:10.1109/ICSE-Companion58688.2023.00091.

[5] G. Li, Y. Tang, X. Zhang, B. Yi, A Deep Learning Based Approach to Detect Code Clones, in: Proceedings of the International Conference on Intelligent Computing and Human-Computer Interaction (ICHCI), 2020, pp. 337–340. doi:10.1109/ICHCI51889.2020.00078.

[6] C. Admiraal, W. van den Brink, M. Gerhold, V. Zaytsev, C. Zubcu, Deriving Modernity Signatures of Codebases with Static Analysis, 2023. doi:10.2139/ssrn.4536605.

[7] E. N. Akimova, A. Y. Bersenev, A. A. Deikov, K. S. Kobylkin, A. V. Konygin, I. P. Mezentsev, V. E. Misilov, PyTraceBugs: A Large Python Code Dataset for Supervised Machine Learning in Software Defect Prediction, in: Proceedings of the 28th Asia-Pacific Software Engineering Conference (APSEC), 2021, pp. 141–151. doi:10.1109/APSEC53868.2021.00022.

[8] A. Alhefdhi, H. K. Dam, H. Hata, A. Ghose, Generating Pseudo-Code from Source Code Using Deep Learning, in: Proceedings of the 25th Australasian Software Engineering Conference (ASWEC), 2018, pp. 21–25. doi:10.1109/ASWEC.2018.00011.

[9] C. Ferretti, M. Saletta, Naturalness in Source Code Summarization. How Significant is it?, in: Proceedings of the 31st IEEE/ACM International Conference on Program Comprehension (ICPC), 2023, pp. 125–134. doi:10.1109/ICPC58990.2023.00027.

[10] A. A. Sawant, P. Devanbu, Naturally! How Breakthroughs in Natural Language Processing Can Dramatically Help Developers, IEEE Software 38 (2021) 118–123. doi:10.1109/MS.2021.3086338.

[11] A. Ezen-Can, A Comparison of LSTM and BERT for Small Corpus, CoRR abs/2009.05451 (2020). URL: https://arxiv.org/abs/2009.05451. arXiv:2009.05451.

[12] H. Yoon, Finding Unexpected Test Accuracy by Cross Validation in Machine, International Journal of Computer Science and Network Security (IJCSNS) 21 (2021) 549–555. doi:10.22937/IJCSNS.2021.21.12.76.

[13] N. Amit, D. G. Feitelson, The Language of Programming: On the Vocabulary of Names, in: Proceedings of the 29th Asia-Pacific Software Engineering Conference (APSEC), 2022, pp. 21–30. doi:10.1109/APSEC57359.2022.00014.