Leveraging deep learning for Python version identification

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

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

Abstract
Python, recognized 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 for a code snippet. We explore the complexities of handling Python data and the DL techniques to achieve high classification accuracy. Our experimental results show, that LSTM with CodeBERT embedding achives an accuracy of 92%. This success can be attributed to the LSTM's proficiency in capturing structural details of the hierarchical nature of a source code, complemented by CodeBERT's ability to discern contextual differences between keywords and variable names. This research provides insights into the challenges associated with utilizing programming languages for deep learning models and suggests potential solutions for addressing these issues. The envisioned applications extend to predicting the minimum required version for individual files or an entire code base.

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. As Python continues to adapt and progress, one critical aspect becomes increasingly pivotal for

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); v.zaytsev@utwente.nl (V. Zaytsev)

both compatibility and ongoing maintenance: the precise determination of the Python version. 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 utilize the functionalities of the version they use. There is a very limited number 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.
The increased utilization 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 summarization [4], and detecting code clones [5]. Deep learning models have an edge over simpler machine learning classification methods in certain tasks due to their ability to learn intricate patterns and representations from data. Their depth allows them to automatically extract hierarchical features, capturing complex relationships that may be challenging for simpler models. Source code often follows a hierarchical structure, with functions, classes, and modules interacting in intricate ways. So, Deep Learning models can understand not just individual lines of code but also the broader context in which they exist.
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. A research can be sumamrized by the following research question:

- Which DL model provides the highest level of accuracy when classifying Python versions?

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, 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 recognizing 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 analyzer [1]. It was developed to assess and quantify the extent to which Python 3 features are utilized, including their adoption rate and the context in which they are applied. At the heart of PyComply lies the foundation of its grammar formalism,

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

which serves to define the Python syntax. Additionally, parser actions have been seamlessly incorporated into this grammar to aid in recognizing 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 would contain 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 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 popular top 50 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 `ast` module [2], we generated Abstract Syntax Trees (ASTs) for each file, discarding unparsable ones. Given that the module relies on abstract grammar for text parsing, it is noteworthy that certain files associated with earlier versions of Python may remain unprocessable. This is attributed to their lack of alignment with the presently specified abstract grammar encapsulated within the package. 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. Oversampling is more beneficial in our case, since some of the classes have a minuscule number of instances. Deep learning methods require a large corpus for their numerous parameters to effectively learn diverse patterns and variations within the data. The abundance of examples facilitates generalization to new instances and mitigates overfitting, enhancing the model's adaptability and robustness in real-world scenarios.

To deal with the mentioned issue, we apply a widely-used approach to synthesizing new class instances called Synthetic Minority Oversampling TEchnique (SMOTE) [9]. The SMOTE algorithm operates by calculating the difference between a sample and its nearest neighbor, multiplying this difference by a random number between 0 and 1, and then adding the result to the sample to generate a new synthetic example in feature space. This process is iteratively applied to the next nearest neighbor until a user-defined number of synthetic examples is generated. Ensuring sufficient data representation in each class, we balanced the dataset by

[2]https://docs.python.org/3/library/ast.html

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	1,200,199	192	13,985	4,719	16,831	14,151	26,685	7,166	20,741	74	514	2,317	425	6,538	25,146	184	722	63	1,792	36

Table 1
Number of instances per class.

	LSTM[10]	TCN[11]	TextCNN[12]	BERT [13]	CodeBERT[14]	XLNet[15]
Word2Vec[16]	🟩	🟩	🟩	🟥	🟥	🟥
CodeBERT[14]	🟩	🟩	🟩	🟩	🟩	🟥
XLNet[15]	🟥	🟥	🟥	🟥	🟥	🟩

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

undersampling classes with more than 4000 instances and generating additional observations using SMOTE for classes with fewer than 4000 instances.

4. Training details

Table 2 presents all the combinations of word embeddings with classifiers, which resulted in 9 models trained for the experiments. Red combinations were excluded due to potential conflicts between word embedding methods and models. For instance, the static nature of Word2Vec may clash with BERT's contextualized embeddings, limiting the latter's effectiveness. Mismatched training objectives and data sources can introduce inconsistencies, leading to larger, more complex models that may impact computational efficiency. Fine-tuning challenges necessitate careful parameter tuning, and the fusion of BERT's task-specific embeddings with Word2Vec may dilute the former after fine-tuning.
We used the `gensim` library for Word2Vec implementation, training the model on a dataset. After experimenting with various embedding sizes, we found 100 to be optimal. Hyperparameters were fine-tuned, setting the window size to 5 and the minimal frequency to 1. For CodeBERT, a pretrained model from CodeSearchNet was used. BERT, CodeBERT, and XLNet had a token limit of 512, so longer files were divided into batches. A batch size of 128 was chosen for a balance between speed and accuracy. XLNet's word embeddings, pretrained on English text, had a parameter set to 128 tokens. The models shared common configurations, including a learning rate of 2e-5 with the Adam optimizer, sparse categorical cross-entropy loss, 768-unit hidden representation, batch size of 64, 100 training epochs, and a final dense layer with sigmoid activation. Attention masks and dropout layers were applied for BERT and CodeBERT, while XLNet used a 0.1 dropout probability. Early stopping, monitoring loss and halting training after 3 epochs without improvement, was implemented to manage computational cost. LSTM, TCN, and TextCNN shared a similar setup with the Adam optimizer, default learning rate, sparse categorical cross-entropy loss, batch size of 64, 100 training epochs, and a final dense layer with softmax activation. They also utilized the same early stopping criteria. The dataset was split

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

Table 3
Results of each model for accuracy, balanced accuracy, precision, recall, and f1-score on the test set of short code snippets.

into a training set and a test set with a 60:40 ratio. Additionally, 20% of the training set was allocated for validation purposes during the training phase to monitor the model's learning progress.

5. 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 categorizing 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 future application of the model, where it can be used to predict the minimal Python version for the file or an entire code base, since a single file, technically, is a combination of a code snippets. So, 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. 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.

6. Discussion

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.
Table 3 presents the results from the evaluation, demonstrating the superiority of the LSTM model with CodeBERT embedding, achieving 93% for each metric. Figure 1 illustrates the F1-scores for individual classes attained by four models, two of which were top performers while the other two performed poorly to show the contrast of results. The findings clearly indicate that replacing Word2Vec with CodeBERT embeddings leads to noticeable improvements

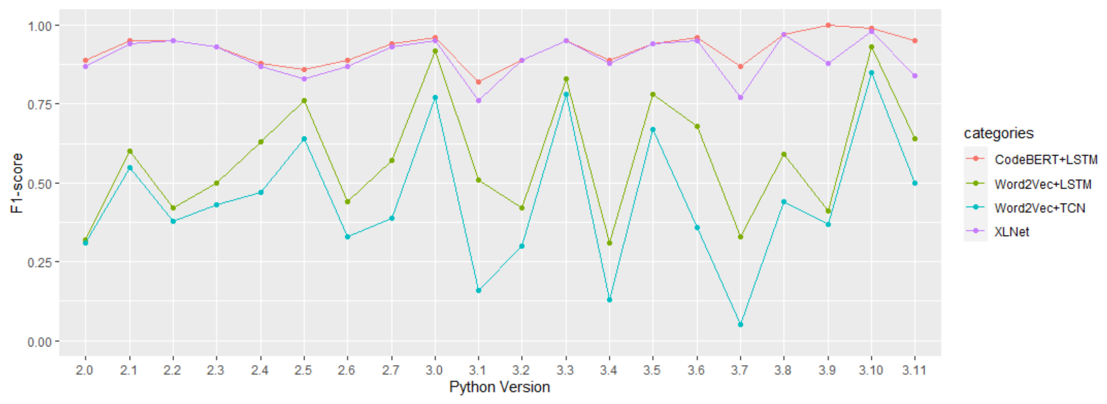

Figure 1: F1 score of four models per each Python version

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 [17]. Coupled with LSTM, this model excels in handling sequential data, enabling it to retain tokens in memory over extended periods, particularly beneficial for programming languages with dependencies throughout the code [18]. An interesting finding reveals lower validation accuracy for transformers compared to LSTM and TCN, consistent with previous research [19]. BERT, in particular, exhibits reduced performance on smaller datasets, due to its original training on extensive corpora. This limitation is emphasized 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 test accuracy. The same observation has been reported previously, suggesting that it may be attributed to the inadequacy of the testing data [20]. Since the model was trained on samples with distinctive version features, test set may contain instances that share structural and lexical similarities with the training data. Similarity arises from consistent function and library names across versions. Despite not being identical, training and test instances exhibit similarity due to the constrained source code vocabulary [21].

One of the many challenges is an infinite vocabulary span, signifying endless possibilities of potential names for identifiers[18, 21]. 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 summarization, where the presence of natural language improved summarization accuracy [17]. 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's 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's 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.

7. Conclusion

We examined nine deep-learning models for Python version classification. LSTM with CodeBERT embedding yielded the highest balanced accuracy of 92% when applied to a dataset containing Python code snippets. This is achieved due to the LSTM's ability to capture structural details within the data, which is beneficial since the source code follows a hierarchical structure. Furthermore, we benefited from the CodeBERT's ability to understant contextual differences between tokens. This is particularly advantageous in Python, where certain keywords can be used as variable names, and the difference depends on the context. Future improvements on the accuracy of the classification can be made via 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. In future applications, the model could serve to predict the minimum required version for an individual file or the entire code base.

References

[1] B. A. Malloy, J. F. Power, Quantifying the Transition from Python 2 to 3: An Empirical Study of Python Applications, in: 2017 ACM/IEEE Int. Symposium on 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: 2023 IEEE/ACM 31st 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: 2022 10th 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: 2023 IEEE/ACM 45th ICSE-Companion, 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: 2020 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, in: Journal of Systems and Software, 2023. doi:http://dx.doi.org/10.2139/ssrn.4536605.

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

[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: 2021 28th 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: 2018 25th ASWEC, 2018, pp. 21–25. doi:10.1109/ASWEC.2018.00011.

[9] N. V. Chawla, K. W. Bowyer, L. O. Hall, W. P. Kegelmeyer, SMOTE: Synthetic Minority over-Sampling Technique, J. Artif. Int. Res. 16 (2002) 321–357.

[10] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural computation 9 (1997) 1735–1780.

[11] S. Bai, J. Z. Kolter, V. Koltun, An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling., CoRR abs/1803.01271 (2018). URL: http://arxiv.org/abs/1803.01271. arXiv:1803.01271.

[12] Y. Kim, Convolutional neural networks for sentence classification, in: Proceedings of the 2014 Conference on EMNLP, Association for Computational Linguistics, Doha, Qatar, 2014, pp. 1746–1751. URL: https://aclanthology.org/D14-1181. doi:10.3115/v1/D14-1181.

[13] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, 2018. URL: http://arxiv.org/abs/1810.04805, cite arxiv:1810.04805Comment: 13 pages.

[14] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, M. Zhou, CodeBERT: A Pre-Trained Model for Programming and Natural Languages, 2020. URL: http://arxiv.org/abs/2002.08155, cite arxiv:2002.08155Comment: Accepted to Findings of EMNLP 2020. 12 pages.

[15] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, Q. V. Le, XLNet: Generalized Autoregressive Pretraining for Language Understanding, in: H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, R. Garnett (Eds.), Advances in Neural Information Processing Systems, volume 32, Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper_files/paper/2019/file/dc6a7e655d7e5840e66733e9ee67cc69-Paper.pdf.

[16] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, J. Dean, Distributed Representations of Words and Phrases and their Compositionality, Advances in Neural Information Processing Systems 26 (2013).

[17] C. Ferretti, M. Saletta, Naturalness in Source Code Summarization. How Significant is it?, in: 2023 IEEE/ACM 31st ICPC, 2023, pp. 125–134. doi:10.1109/ICPC58990.2023.00027.

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

[19] 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.

[20] H. Yoon, Finding Unexpected Test Accuracy by Cross Validation in Machine, IJCSNS 21 (2021) 549–555. doi:https://doi.org/10.22937/IJCSNS.2021.21.12.76.

[21] N. Amit, D. G. Feitelson, The Language of Programming: On the Vocabulary of Names, in: 2022 29th APSEC, 2022, pp. 21–30. doi:10.1109/APSEC57359.2022.00014.