# The Limits of the Identifiable: Challenges in Python Version Identification with Deep Learning

Marcus Gerhold
*Formal Methods & Tools*
*University of Twente*
Enschede, the Netherlands
m.gerhold@utwente.nl

Lola Solovyeva
*Computer Science*
*University of Twente*
Enschede, the Netherlands
o.solovyeva@student.utwente.nl

Vadim Zaytsev
*Formal Methods & Tools*
*University of Twente*
Enschede, the Netherlands
vadim@grammarware.net

***Abstract*—The evolution of Python requires accurate version identification to facilitate compatibility and ongoing support. We extend previous work on deep learning models for Python version identification, where LSTM and CodeBERT achieved a 92% accuracy on short code snippets. We further expand these results to larger realistic files, utilising code segmentation techniques for varying input granularities. These techniques ranged from per-line analysis to larger code segments. Our findings show that while LSTM with CodeBERT embeddings maintained high accuracy on short snippets, performance significantly drops on longer segments, particularly in balancing information retention and misclassification risks. Notably, `import`-statement analysis, despite being the most intuitive indicator of version requirements, reached only a 30% accuracy. This exposes the limitations of our approach when encountering rare or user-defined modules. The findings expose the limitations of deep learning for language version identification, and suggest that alternative approaches may be necessary for high accuracy on larger datasets.**

***Index Terms*—software language identification, deep learning, Python, CodeBERT**

## I. Introduction

Language identification is an innovative concept in software engineering that addresses issues of growing importance like code compatibility, reverse engineering of legacy code bases, or IDE support such as syntax highlighting or code completion [1]. As a programming language continues to mature, the role of accurate identification shifts from a beneficial addition to a critical necessity since preciseness is essential for facilitating maintenance, ensuring robustness, and supporting seamless integration of diverse software systems [2].

In the landscape of evolving languages, Python stands out due to its long history, notable version disparities, and widespread usage [3], [4]. At the time of writing, it had 20 supported minor versions. The most significant divide between the 2.X and 3.X versions marks a distinct evolutionary jump in the language's syntax and functionalities. Critically, this makes projects developed in one of the versions incompatible with the respective other. This incompatibility exemplifies the decisive role of accurate language identification in maintaining compatibility within Python's evolving ecosystem.

Despite the need for precise identification in Python, contemporary approaches reveal shortcomings, particularly when applied to real-world, diverse codebases [5]. Pragmatic tools like Vermin [6] struggle with accurately discerning between minor Python versions, especially in mixed-version environments, where subtle syntax and feature variations complicate detection [7]. As software projects grow in scale, these discrepancies escalate, which emphasises the need for advanced solutions to effectively address them.

At the same time, deep learning techniques enjoy increased successes and have become more prominent in the field of software engineering. For instance, deep learning has proven highly beneficial in tasks such as detecting code clones [8], identifying code smells [9], and code summarisation [10]. Hence, a very natural conclusion involves using deep learning for language identification; and most prominently in addressing Python's minor and major version discrepancies.

This paper extends our earlier work [11] in which we determined the most accurate deep learning model for classifying Python versions in short code snippets. Our experiments showed the combination of CodeBERT and LSTM to be the most accurate with 92% [11]. As an intuitive progression, we apply these findings to expanded experiments involving complete Python files and projects. These experiments utilise techniques of varying granularity for segmentation, encompassing approaches such as analysing the first $n$ lines, `import` statements, and abstract syntax tree (AST) nodes.

Contrary to expectations, our experiments show that none of the text segmentation techniques applied in this context achieved more than 30% accuracy. This reveals a substantial performance gap in deep learning for Python version identification: while effective for short snippets, the accuracy degrades as file size grows. More importantly, this exposes the inadequacy of current deep learning techniques for effective language identification in complex, real-world software environments. Simultaneously, this demands the need for advancements in either deep learning for language identification or the exploration of alternative techniques.

*Overview of the paper.* We present related work in section II and briefly recall results of our earlier work in section III. In section IV we provide an overview of our approach, and expound classifiers and text segmentation techniques. Our experimental setup is outlined in section V and results are presented in section VI. We conclude with closing remarks and give directions for future research in section VIII.

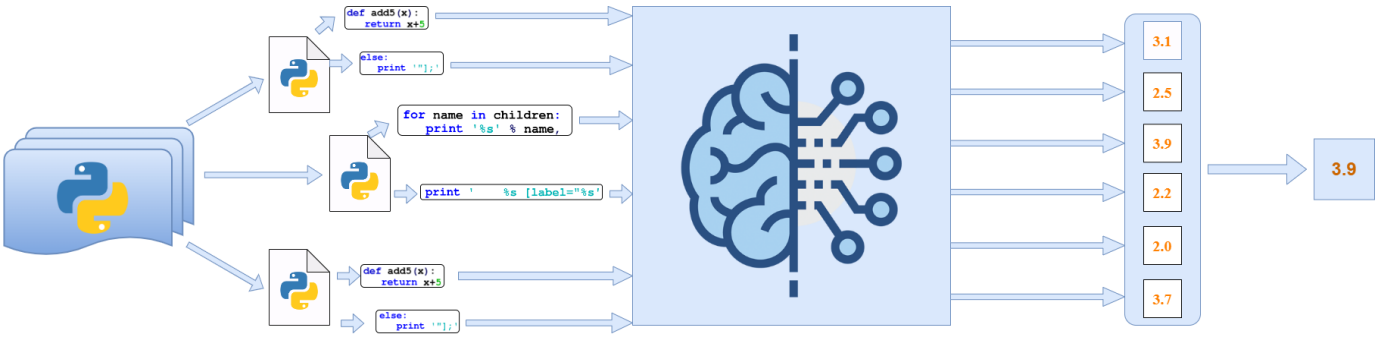


Fig. 1. Research pipeline for finding minimal version for a Python project using a deep neural network.

## II. Related Work

Software language identification [1] has been an acknowledged problem at least since 1973 when Unix Research Version 4 included a tool called `file`. Refining solutions to detect dialects and versions within a language has far reaching consequences for languages like COBOL or SQL with hundreds of dialects in active simultaneous use [12]. However, there has been limited attention and research dedicated to the problem of identifying Python versions of a file or a project. An existing tool, known as Vermin [6], 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 [13]. Additionally, there is a Chrome extension named PyVerDetector [14], which empowers users to select a specific Python version and validate the compatibility of code snippets on Stack Overflow. 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 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 [5], which is a Python compliance analyser. 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. PyComply's algorithm follows a grammar which defines the Python syntax and is annotated with semantic actions recognising distinctive features of Python 3.

Looking broader, the concept of *pythonicity* has been investigated on several occasions, and even its definition has evolved far beyond the original "beautiful is better than ugly" idea towards a collection of community-accepted idiomatic coding conventions. Alexandru et al. compared its perception by junior and senior developers, and found significant differences [15]. Leelaprute et al. demonstrated that pythonic code is more efficient in terms of both memory use and execution time [16]. Phan-udom et al. built a tool to suggest pythonic idioms during code review [17]. Zhang et al. built another tool to rewrite non-idiomatic code towards increasing its pythonicity, and successfully submitted a number of pull requests in real projects with it [18]. Sakulniwat et al. [19], Farooq et al. [20] and later Admiraal et al. [13] experimented with different ways of visualising adoption of idiomatic pythonic code over time.

Deep learning techniques have also been applied before to Python data for various reasons. Akimova et al. [21] 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. [22] 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. Sandouka et al. [23] proposed a dataset of Python code with smells, and applied machine learning to detect Long Method and other less prominent smells. Chen et al. [24] before them also found (with non-ML static analysis) that Long Method and Large Class are the most popular smells. More precisely, as reported by Vavrová et al., Long Method occurs twice as often in Python code as it would in Java code, while some other smells like Long Parameter List, are seven times less likely to occur [25]. This entire body of knowledge shows that research results from other software languages cannot be simply assumed to be applicable to Python, which has its own distribution of smells, its own adoption pace and strategies, and its own ways of reaching community consensus.

## III. Background

In earlier research [11], we compared nine distinct deep learning models to develop a classifier capable of identifying subtle distinctions between Python versions. The objective was to input a short Python code snippet and have the classifier determine the minimum required version for its execution. In this section, we contextualise our prior research in order to justify specific technical choices made in this paper.

### A. Python Classifiers

Table I demonstrates considered models and techniques for text embedding. Our selection is based on prior successes of deep learning in the software engineering domain [8]–[10]. Given Python's reputation as one of the most easily readable

TABLE I
COMBINATIONS OF WORD EMBEDDINGS AND CLASSIFIERS.

| | Word2Vec | CodeBERT | XLNet |
|---|---|---|---|
| **LSTM** | ✓ | ✓ | |
| **TCN** | ✓ | ✓ | |
| **TextCNN** | ✓ | ✓ | |
| **BERT** | | ✓ | |
| **CodeBERT** | | ✓ | |
| **XLNet** | | | ✓ |

programming languages, closely resembling the structure of a natural language, models that were pre-trained on natural language were also deemed suitable for experiments [26]. As evident from the ticks, not all combinations were considered. This was done to avoid potential conflicts arising from the interplay between word embedding methods and models. For example, the static nature of Word2Vec may conflict with the contextualised embeddings of BERT, thereby restricting the effectiveness of the latter. Inconsistencies can arise from mismatched training objectives and data sources, resulting in larger and more complex models that could impact computational efficiency. Fine-tuning poses challenges, requiring meticulous parameter tuning, and the integration of BERT's task-specific embeddings with Word2Vec may lead to a dilution of the former after fine-tuning.

The final models were trained on a Python dataset available at https://github.com/LolaSolovyeva/SLI that consisted of short code snippets with their corresponding minimal version required for their execution. An example of a short code snippet is provided on Figure 2.

The dataset displayed a notable imbalance, primarily attributed to variations in the usage frequency among different Python versions, with certain versions being more commonly utilised than others. The distribution of the classes and their observations can be found in Table III. Nevertheless, the imbalance problem was solved by synthesising new observations for minority classes using Synthetic Minority Oversampling TEchnique (SMOTE) [27]. Each class comprised 4,000 samples, resulting in a training dataset consisting of a total of 80,000 Python code snippets.

```
if transactions:
      Transaction.create_transactions()
node.generate_emptyState()

S.initial_events()
while not queue.isEmpty() and clock <= targetTime:
      next_e = queue.get_next_event()
      clock = next_e.time
      Event.execute_event(next_e)
      Queue.remove_event(next_e)

print results
```

Fig. 2. A short code snippet example

### B. Overview of the Results

The results of the study are summarised in Table II. We demonstrated that LSTM with CodeBERT embedding yielded the highest balanced accuracy of 92% and F1-score of 93% when applied to a test set containing Python code snippets. The results clearly demonstrate that substituting Word2Vec with CodeBERT embeddings yields significant enhancements, as measured by balanced accuracy and F1-score. This illustrates that the utilisation of domain-specific embeddings such as CodeBERT markedly improves performance of the model in classifying instances across all categories. Proficiency of CodeBERT in grasping contextual nuances and capturing distant token relationships proves crucial in the context of structured text, such as source code [28]. When combined with LSTM, this model excels in processing sequential data, allowing it to retain tokens in memory for extended periods. This capability is particularly advantageous for programming languages characterised by dependencies distributed throughout the code [29].

## IV. OUR APPROACH

This section presents our approach in details and provides the overview of the pipeline illustrated in Figure 1, insights into the used classifier and text segmentation techniques.

### A. Overview

Given the satisfactory performance of the classifier on short code snippets, our goal is to extend its application to files, which are essentially a collection of code snippets. In addition to employing a high-performance classifier, it is imperative to identify an optimal text segmentation technique. This is essential to ensure the preservation of a high level of precision in determining the resulting minimal required version.

Figure 1 outlines the essential steps required to achieve the ultimate objective of determining the minimum required Python version. In the case, where we work with an entire code base, it is necessary to partition it into individual files. The justification lies in the dependency of the minimal required version for a code base on the minimal required version of all the files within that code base. Subsequently, text segmentation is employed for each file. Considering that the most effective model has an input limit of 512 tokens and was trained on short code snippets, we must identify an optimal approach to divide a file into segments. Various text segmentation methods are proposed in subsection IV-C and assessed in section VII. Once each file is segmented, the individual segments are input into the classifier, providing the minimum Python version required for each snippet's execution. The classifier outputs are then aggregated into a vector, with the vector's length corresponding to the number of segments in each file within the code base. The ultimate minimal required Python version for the code base is determined as the maximum version output by the classifier in the vector.

The pipeline mirrors a standard procedure applied by developers that try to identify the language version required for the program's execution: Determining the required language

TABLE II
RESULTS OF EACH MODEL FOR ACCURACY, BALANCED ACCURACY, PRECISION, RECALL, F1-SCORE ON THE TEST SET OF SHORT CODE SNIPPETS [11].

| 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 |

version from a given code involves examining syntax features, library compatibility, language constructs, deprecation warnings, print statements, and third-party tool usage. Typically, developers conduct a thorough analysis of these factors by inspecting the code file and drawing upon their existing knowledge. Consequently, the automated pipeline emulates this standard approach and streamlines the process through automation. This ensures an efficient and accurate identification of the minimal required language version, maintaining compatibility and adherence to language-specific features.

### B. Used Classifiers

The classifier used in this study originated from earlier research [11]. There our objective was to train a model capable of predicting the minimum required version for a given Python code snippet. The combination of CodeBERT and LSTM proved to be the most accurate with 92% on short code snippets, cf. Table II. We briefly recall both here.

*a) CodeBERT [30]:* It is a pre-trained model that is designed for both programming language and natural language processing tasks. It is capable of learning general-purpose representations that can be applied to a wide range of downstream NL-PL applications, such as natural language code search and code documentation generation. CodeBERT is built with a neural architecture that utilises the Transformer model, and it is trained with a hybrid objective function that includes a pre-training task for replaced token detection. This task involves identifying suitable alternatives for tokens, which are sampled from generators. By doing this, CodeBERT can leverage both bimodal data of NL-PL pairs and unimodal data, where the former provides input tokens for model training, while the latter aids in the learning of better generators.

In our context, we use the weights from the model for the embedding of the data. It is not a surprise that the embedding demonstrated great results, since CodeBERT possesses unique advantages that could significantly contribute to our objectives. For example, CodeBERT exclusively utilises raw textual information, distinguishing it from models that depend on additional data such as Abstract Syntax Trees (ASTs) and require the transformation of input code into ASTs. This distinctive feature enables the application of CodeBERT to individual lines of source code, thereby easing the constraint associated with line-level parsability [31]. It is openly available via the repository [32], allowing for straightforward utilisation.

*b) LSTM [33]:* Long Short-Term Memory is a specialised type of recurrent neural network designed to process sequences by using memory cells and gates to manage information flow. It addresses the vanishing gradient problem, allowing it to capture long-range dependencies and patterns in data. LSTMs are widely used in tasks involving sequential data, such as language modelling and time series prediction, though their complexity has led to the development of more advanced architectures like transformers.

In our context, LSTMs are well-suited for understanding the intricate structures of source code, as they can learn hierarchical representations, automatically extract relevant features, and consider the entire context of a code sequence. Various studies in the software engineering domain show that LSTM succeeds in source code segmentation [34], code smell prediction [9] and code clone detection [35].

TABLE III
NUMBER OF INSTANCES PER VERSION FOR TRAINING A PYTHON CLASSIFIER ON SHORT AND LONG CODE SNIPPETS.

| **Version** | **2.0** | **2.1** | **2.2** | **2.3** | **2.4** | **2.5** |
|---|---|---|---|---|---|---|
| **Short** | 1200199 | 192 | 13985 | 4719 | 16831 | 14151 |
| **Long** | 7878 | 598 | 1039 | 1266 | 3311 | 5099 |
| **Version** | **2.6** | **2.7** | | | | |
| **Short** | 26685 | 7166 | | | | |
| **Long** | 14141 | 3243 | | | | |
| **Version** | **3.0** | **3.1** | **3.2** | **3.3** | **3.4** | **3.5** |
| **Short** | 20741 | 74 | 514 | 2357 | 425 | 6538 |
| **Long** | 0 | 0 | 3122 | 1422 | 1264 | 2621 |
| **Version** | **3.6** | **3.7** | **3.8** | **3.9** | **3.10** | **3.11** |
| **Short** | 25146 | 184 | 722 | 63 | 1792 | 36 |
| **Long** | 5987 | 763 | 346 | 53 | 0 | 0 |

### C. Text segmentation

Given that the model undergoes training on short code snippets, it becomes imperative to partition the file into segments. The number of segments should be enough to draw an accurate conclusion about the minimal required version of the Python file. An extensive number of segments could potentially cause an escalation in the probability of errors. The following paragraphs briefly describe the methods employed in this research.

*a) First $n$ words:* This method proposes to truncate the file to the first $n$ words, where $n$ is set to the number that increases the accuracy of classification. A more detailed

investigation into the potential equivalence of $n$ is expounded upon in the subsequent sections.

*b) Per part:* This method splits the file into parts with the length $n$. This approach enables us to make decisions based on an entire file, as unique features may emerge at any point within the file.

*c) Per line:* This method suggests splitting the file each time a new line is introduced. The rationale behind this strategy can be expounded in light of the model's training regimen since it is trained on short code snippets, that can be conceptualised analogously to singular lines of code.

*d) `import` statements:* It allows a user to leverage other existing libraries. On occasion, those libraries can serve as strong indicators of the required Python version to compile the file. This approach tests if the minimal version can be solely derived from the imported libraries.

*e) AST nodes:* The version of a file can be changed due to the use of a certain feature that can be architecturally different from its previous interpretation. To capture such a difference, a file should be divided into discrete code fragments predicated upon its abstract syntax tree (AST). Hence, this segmentation entails the extraction of textual content from certain nodes within the file.

## V. Experimental setup

This section presents the technical details behind the conducted experiments. We start by describing the dataset and techniques for its pre-processing, followed by the training details of the classifier and strategies for the evaluation of the performance.

### A. Corpus construction and pre-processing

There is no existing corpus that would solely contain various Python files mapped to their corresponding minimal required version. Hence, there is a need to create a dataset, that consists of code examples for each of the Python versions to perform our experiments.

PyPI [36], the official third-party software repository for Python, provides access to over 450,000 Python packages. We leverage PyPI to extract Python projects, which are subsequently divided into individual files for dataset creation. We acquired the releases of the top 50 Python projects, determined by their popularity as of 15 May 2023. The rationale behind obtaining all releases for each project is to ensure a broader representation of Python versions for analysis. For example, one of the used packages in our dataset is `scipy` [37]. Its initial release is 0.10.0 and required Python 2.6 and one of the latest releases 1.9.3 requires Python 3.8. So by obtaining both releases of the same project, we capture an observation for two distinct classes.

Various methods can determine the minimal required Python version for a project or file. Utilising PyPI as our data source grants access to a project's JSON file, but manual input by developers can introduce inconsistencies. This discrepancy implies that while the Python version may be specified as 3.10, the required version could be designated as 3.8. Given Python's lack of backward compatibility, deploying a package developed with version 3.10 on a 3.8 environment would render it incompatible. In instances where compatibility is maintained, the newly introduced features in 3.10 remain unused, establishing 3.8 as the minimum required version. The versions specified in the JSON file pertain to the entire project. Given that we partition the project into discrete files, the required version for each individual file may diverge, particularly if none of the distinctive features of the specified minimal version are employed. Consequently, we need to establish a methodology for ascertaining the minimum version requirement on a per-file basis. Vermin [6] is employed to establish per-file minimum required versions, as it distinguishes between minor Python versions. Python files are extracted, excluding non-Python files and removing comments to reduce noise. Vermin is then used to obtain the minimal Python version for each file, resulting in a dataset mapping Python files to their respective versions.

Table III presents the number of instances for each class. Evidently the dataset exhibits an imbalance due to variations in the prevalence of certain Python versions over others. This imbalance is taken into account during performance evaluation. Furthermore, we could not capture observations for some of the classes, so those versions are removed from the testing phase.

### B. Training details

While the training of the classifier was conducted in earlier research [11], we expound upon its technical details to ensure the replicability and reproducibility of the obtained results.

To generate CodeBERT embeddings, we utilised weights from a model trained on bi-modal data sourced from CodeSearchNet [32]. Given that BERT cannot handle text longer than 512 tokens, the same limitation applies to CodeBERT. Consequently, we divided the lengthy text into batches, experimenting with batch sizes of 64, 128, 256, and 512 tokens. Ultimately, we settled on a batch size of 128 tokens, striking a balance between speed and performance accuracy. Besides that, special tokens [CLS] and [SEP] must be included in the input, where [CLS] indicates the start of the text and [SEP] acts as a separator between two sentences.

LSTM uses a learning rate of $2e^{-5}$ with Adam optimiser, employs sparse categorical cross-entropy loss, consists of two layers with 100 nodes, uses a batch size of 64, trains for 100 epochs, and incorporates a final dense layer with a softmax activation function. To mitigate the computational cost of training for 100 epochs, early stopping is implemented, which monitors the loss and stops training if no improvement is observed after 3 epochs. The model has halted its training after 47 epochs.

### C. Performance evaluation

Our final objective is to assess the performance of the approach employed in this study. The primary goal is to assess the efficacy of our approach in determining the minimum required Python version for a specified file or an entire code

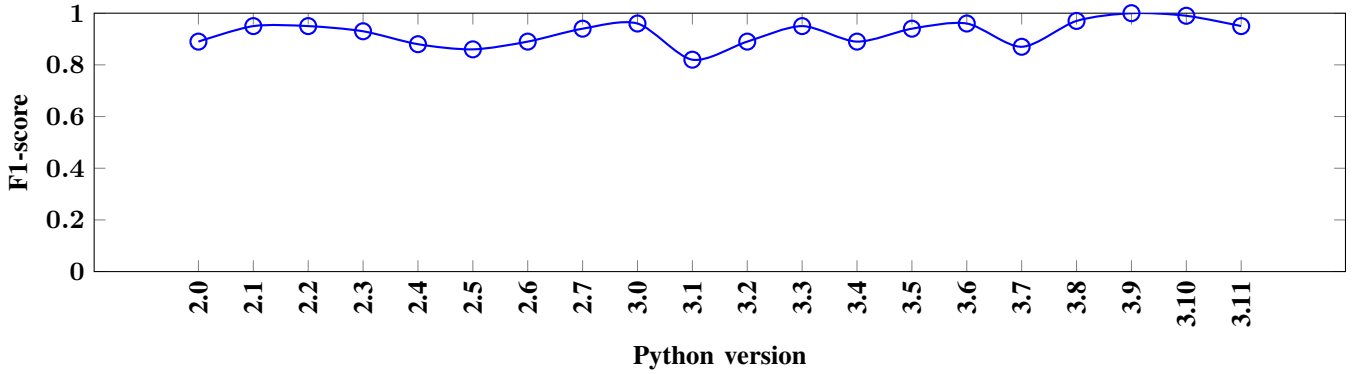


Fig. 3. F1-score for each Python version achieved with the best performing model CodeBERT+LSTM.

base. Furthermore, we explore various segmentation techniques to identify the most effective approach for achieving precise file classification based on the ground truth labels.

The procedural steps for assessing our approach are as follows: Initially, a singular Python file is partitioned into segments using one of the established text segmentation techniques mentioned in subsection IV-C. Subsequently, the model is defined and loaded from the saved checkpoint. Each segment undergoes embedding with CodeBERT before being input into our classifier. The resultant outputs are aggregated into a vector, with dimensions equivalent to the number of segments. The ultimate output is represented by the highest predicted version stored within this vector. It must be the highest, since Python lacks backward compatibility. This implies that the minimal version in the vector would not be compatible with later versions. We use some of the most common metrics to evaluate the performance of text classifiers: accuracy, F1-score and confusion matrix. Since accuracy is useful for balanced datasets but can be misleading in imbalanced datasets, we employ *balanced accuracy*, which is a variation that takes into account the class distribution in the dataset. For a multiclass problem, it is an average of recalls per class [38].

$$\textit{Balanced Accuracy} = \frac{1}{2}\left(\frac{TP}{TP+FN} + \frac{TN}{TN+FP}\right) \quad (1)$$

Precision measures the accuracy of positive predictions, while recall measures the ability of the classifier to identify positive instances correctly. To find a balance between the two metrics, we use the F1-score: the fact that it considers false positives and false negatives, makes it more suitable for imbalanced datasets.

## VI. Results

This section demonstrates the outcomes obtained through the implementation of our methodology on the test set. Our primary objective was to assess the feasibility of the pipeline and identify a text segmentation method that would optimise accuracy, which could be measured with the test set. Despite the previous demonstrations of the outcomes achieved by the best-performing model, we revisit these results to enhance the overall coherence of the study.

### A. Results on short code snippets

Nine distinct models were compared for accuracy, balanced accuracy, the precision, recall, and F1-score as can be observed from Table II. The lowest accuracy, at 51%, is seen when TCN is combined with Word2Vec embedding. In contrast, the highest accuracy, reaching 93%, is achieved when the LSTM model utilises CodeBERT embedding. The same tendency seems to appear with regard to balanced accuracy with the lowest being 42% and highest 92% corresponding to the previously mentioned models. It is worth noting that the LSTM model, when coupled with CodeBERT embedding, achieved the top scores of 93% for each metric. Conversely, the TCN model utilising Word2Vec embedding recorded the lowest scores of 55% across all metrics.

Figure 3 illustrates the F1-scores for individual classes attained by CodeBERT+LSTM. Evident from the graph, the model demonstrates proficient performance across the various classes. However, it encounters some challenges, particularly with versions 2.5, 3.1, and 3.7.

### B. Results on lengthy files

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. We start with splitting the file into segments, using each of the techniques mentioned in subsection IV-C, which is followed by applying the model to each of the segments. The ultimate minimal version is determined as the highest predicted version by the model across all segments. Additionally, it is worth

TABLE IV
Presents the results of each text segmentation technique according to the chosen performance metrics.

| Method | Accuracy | F1-score | Balanced Accuracy |
|---|---|---|---|
| First 128 tokens | 0.27 | 0.23 | 0.25 |
| **`import` statements** | **0.30** | **0.28** | **0.29** |
| Split in parts | 0.09 | 0.05 | 0.08 |
| AST nodes | 0.09 | 0.05 | 0.08 |
| Per line | 0.07 | 0.04 | 0.05 |

noting that versions 3.0, 3.1, 3.10, and 3.11 were excluded from this experiment due to insufficient availability of lengthy files in the scraped data for those versions.

Figure 4 presents the results from parameter tuning for token input into the model. We downsampled the dataset, acquiring the most representative instances, to reduce computational costs and oversampling to noise. Achieving a nearly equal number of instances per class, we selected accuracy as the tuning metric for the parameter. It is evident that initially, as the number of tokens increased, so did the accuracy. However, the highest point of 25% was achieved when using 128 tokens, after which the accuracy began to decline. Therefore, 128 token input was used throughout the further experiments for other text segmentation techniques that required the input shape such as splitting the file into parts.

Table IV demonstrates obtained results across the metrics chosen for the evaluation of the performance. It is apparent that the most favourable outcomes were attained through the utilisation of `import` statements, yielding accuracy, F1-score, and balanced accuracy rates of 30%, 28%, and 29%, respectively. The second favourable approach involves the initial 128 tokens, yielding results nearly as good as those derived from `import` statements. The remaining methods exhibited a significant deficiency in performance.

Figure 5 represents the achieved accuracy per class by applying the LSTM model with CodeBERT embedding on lengthy files that were segmented using previously mentioned methods. It can be seen that the most successful attempt was achieved by applying the model to `import` statements from the file reaching 30%. Even though the method did not show the highest results per class, it did not experience a significant drop in the accuracy of the classification in comparison to other methods. Furthermore, the figure illustrates that techniques extracting the initial 128 tokens and `import` statements demonstrate greater success in Python V2 prediction. Conversely, in Python V3, dividing files by lines or segmenting them into parts of size $n$, where $n$ is 128, outperforms other methods.

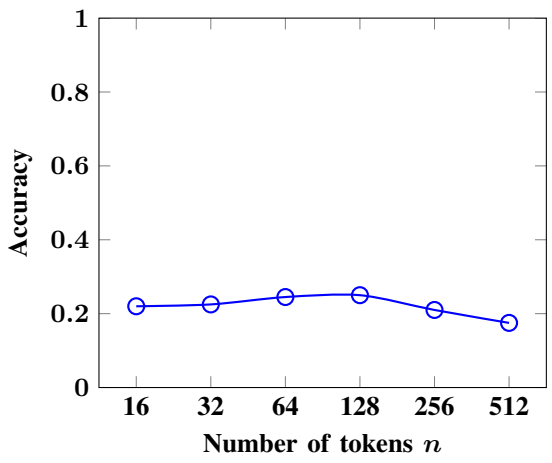


Fig. 4. The accuracy on lengthy files shortened to $n$ first words, where $n$ is a number of tokens. We used CodeBERT embedding with the LSTM model since it demonstrates the highest performance.

We can observe that certain classes attain a notable classification accuracy, like achieving over 75% accuracy for version 3.7 through per-line segmentation. Conversely, there are classes, such as 3.1, that exhibit less promising outcomes, with approximately 5% accuracy when employing file partitioning.

## VII. Discussion

In the following section, we illuminate key findings derived from the experimental outcomes and elucidate the challenges and constraints associated with handling Python data.

### A. Highlights on text segmentation

Since BERT models have a maximum token limit, which is typically 512 tokens for the original BERT architecture, this means that we need to either truncate the file or split it into similar segments. However, the chosen method for file segmentation can influence the final result of its classification. We tested five techniques for segmenting the file, aiming to find the method that maximises the accurate classification.

Predicting the minimal Python version by analysing the `import` statements in the file has proven to achieve the highest accuracy out of all the methods used in this study. This method seems intuitive since `import` statements list libraries or modules that would carry specific version requirements. Hence, `import` statements would be a clear indicator of the minimum required version for the file, unless there are some syntactic features present indicating a later version. Nevertheless, even though this method is the most successful in this study, it could not reach an accuracy higher than 30%. This can be attributed to the fact that the model's training data encompasses more than just `import` statements, potentially resulting in its lack of awareness regarding all libraries. This also holds true for imported modules that users may have personally developed. If a user has created a module named $X$ with specific version dependencies, the model may not recognise that importing this module necessitates the same version. This is likely due to the rare appearance of this module in the training corpus, making it challenging for the model to learn these nuances. Furthermore, there could be cases where the file does not have any `import` statements, so the model must only rely on the syntactic features that make the version distinguishable from the others.

The second most successful method has shown to be a simple file truncation to the first 128 words, which presented an accuracy of 27%. A lower accuracy can be justified by eliminating a chunk of the file that can contain crucial information for the correct classification since we only consider the beginning of it. Nevertheless, one could argue the reason why this method secured the second position. Given that `import` statements typically appear at the start of the file, they are part of the first 128 words, influencing the results of the method. Hence, this highlights the significance of drawing conclusions about the minimal version based on `import` statements.

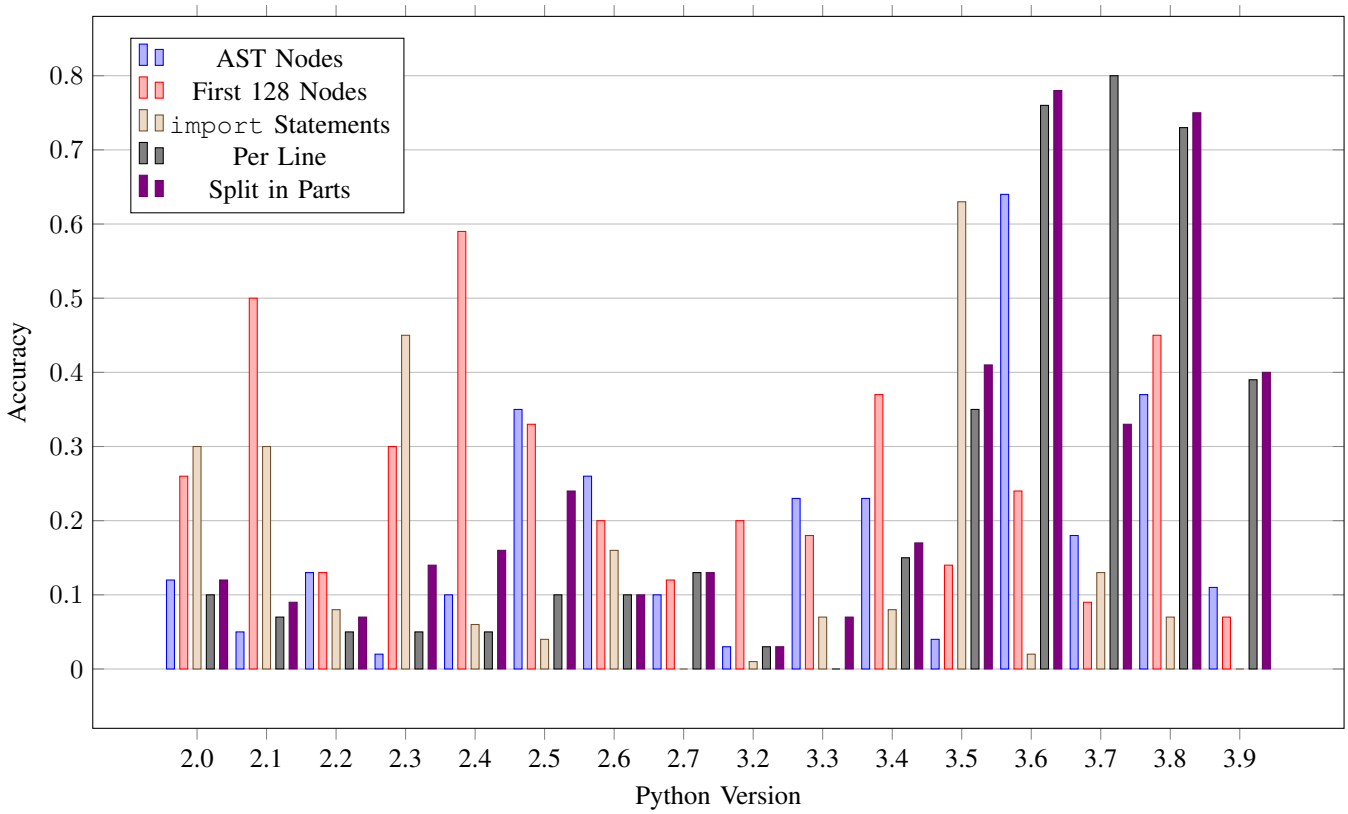


Fig. 5. The accuracy of prediction for each class by LSTM model with CodeBERT embedding using various text segmentation techniques listed in the legend.

Utilising the model on file chunks did not yield favourable results. The primary challenge lies in striking a balance between offering the model sufficient information for accurate classification and minimising the potential for misclassification. The greater the number of chunks we input, the greater the chances of the model misclassifying one of them. It is sufficient for just one chunk to be misclassified to lead to an incorrect assignment of the entire file to the wrong minimal version. So, lower accuracy for the other three methods can be explained by the reason above.

A noteworthy observation pertains to the superior performance exhibited by the initial 128 tokens and `import` statements relative to other methods in files authored in Python 2. Conversely, a reversal of trends was observed in the context of Python 3. This suggests that distinguishing between Python 3 versions is facilitated by the file syntax alone, while the syntax of Python 2 presents increased complexity in differentiation.

Unfavourable results may stem from ability of Vermin in accurately identifying the minimal required version. This concern is particularly significant as the tool labels data as ground truth, and inaccuracies in Vermin could propagate into misleading model results. Bugs with 169 cases of incorrect version identifications were previously reported [7]. A study [13] found that a non-negligible fraction of tests fail on the oldest and newest Python versions, indicating Vermin's lack of awareness of all features in those versions. This leads to poor generalisability and introduces bias in labeled files.

### B. Highlights on working with Python data

There are similarities between the analysis of natural language and the source code since both should be comprehensive and clear for the intended audience. This is essential because a programming language serves as a means of communication between the user and the machine [29]. Especially in the case of Python, which appears to be more readable and English-like in comparison to other programming languages. Nevertheless, the classification of Python files still presented multiple difficulties.

One of the many challenges is an infinite vocabulary span, signifying endless possibilities of potential names for identifiers [29], [39]. 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.

The user incorporates natural language not just when naming variables but also within string variables, print statements, and error messages. The introduction of any form of natural language within the source code can significantly impact the model's performance. This assertion finds support in a study that concentrated on code summarisation. In their research, the presence of natural language in the source code proved advantageous, enhancing the accuracy of summarisation [28]. This can be explained that a condition to obtain a good summary is to have some natural evidence in the source

code such as doc-strings, comments, variable names, etc. Conversely, in our situation, it introduces unwanted noise, detrimentally affecting performance. The goal is to distinguish between the versions, a task achievable solely by identifying unique characteristics associated with each Python version. Therefore, it is essential to isolate these distinctive features from any noise to ensure precise 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` [40] construct 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. Despite an insufficient quantity of extensive files for the inclusion of Python version 3.10 in the experiments, a manual inspection of the scraped files was conducted, revealing the aforementioned scenario. 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 rely on an attention mechanism that allows them to grasp syntactic relationships between tokens. However, BERT-like transformers have constraints on their input capacity, which forces us to either truncate or divide the file into segments. The problem here is that if the file is divided at the wrong point, it can disrupt the integrity of the unique feature's structure, potentially resulting in misclassification since the context is lost.

### C. Limitations

The primary limitation in this study revolves around the reliability of the minimal required Python version, determined through Vermin. Given the absence of a definitive ground truth, reliance on external tools like Vermin, which are not infallible, becomes necessary. A previous research study [13] that employed Vermin, provided some test cases demonstrating its validity, noting that most tests fail for the oldest and newest Python versions. This outcome is anticipated, considering that Vermin's latest update may lag behind the most recent Python release. A noticeable gap exists in the coverage of Python 3.0, stemming from inadequate documentation that fails to mention major version changes. The only solid alternative known in the context of modernity analysis is to rely on an annotated grammar of the language [41].

Another limitation is a possible overlap between the training and testing data. A recent research [42] has indicated that software exhibits similarities to natural languages, implying that source code often displays high levels of repetition and predictability. Additional investigations have highlighted the use of naturalness as an indicator of code quality, specifically in identifying potentially buggy code. If the code exhibits repetitions, some training data might be leaked to the testing data, which would demonstrate high performance that is not actual. Furthermore, as it was demonstrated some Python version are more commonly used, which results in insufficient data for the minority classes.

## VIII. Conclusion

In this study, we explored the feasibility of the proposed pipeline for predicting the minimum required Python version. Additionally, we assessed five text segmentation techniques to discover an optimal balance between providing sufficient information to the classifier and minimising misclassification rates. `import` statements emerged as the most effective technique for capturing information about the required file version. However, classifying extensive files using deep learning poses challenges due to issues such as input limitations, overlap between new and old versions, the presence of natural language in code, and the wide variability in variable names. Even the best performing technique could not achieve accuracy higher than 30%, which can not be considered acceptable. Future enhancements involve masking natural language in the code to reduce data noise and identifying suitable alternatives for unseen variable names, enabling the model to make more accurate predictions based on its training data.

## References


[1] J. Kennedy van Dam and V. Zaytsev, "Software Language Identification with Natural Language Classifiers," in *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering: the Early Research Achievements track (SANER ERA)*. IEEE, 2016, pp. 624–628. [Online]. Available: https://doi.org/10.1109/SANER.2016.92

[2] M. M. Lehman, "On Understanding Laws, Evolution, and Conservation in the Large-Program Life Cycle," *Journal of System and Software*, vol. 1, pp. 213–221, 1980. [Online]. Available: https://doi.org/10.1016/0164-1212(79)90022-0

[3] TIOBE, "TIOBE index," https://www.tiobe.com/tiobe-index/, Nov 2023.

[4] S. Cass, "The Top Programming Languages 2023," *IEEE Spectrum*, Aug. 2023. [Online]. Available: https://spectrum.ieee.org/the-top-programming-languages-2023

[5] B. A. Malloy and 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)*. IEEE Computer Society, 2017, pp. 314–323. [Online]. Available: https://doi.org/10.1109/ESEM.2017.45

[6] M. Kristensen, "Vermin," https://pypi.org/project/vermin, mar 2018.

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

[8] G. Li, Y. Tang, X. Zhang, and 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. [Online]. Available: https://doi.org/10.1109/ICHCI51889.2020.00078

[9] S. Tarwani and 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. [Online]. Available: https://doi.org/10.1109/ICRITO56286.2022.9965048

[10] T. Zhu, Z. Li, M. Pan, C. Shi, T. Zhang, Y. Pei, and X. Li, "Revisiting Information Retrieval and Deep Learning Approaches for Code Summarization," in *Companion Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 328–329. [Online]. Available: https://doi.org/10.1109/ICSE-COMPANION58688.2023.00091

[11] M. Gerhold, L. Solovyeva, and V. Zaytsev, "Leveraging Deep Learning for Python Version Identification," in *Proceedings of the 22nd Belgium-Netherlands Software Evolution Workshop (BENEVOL)*, ser. CEUR Workshop Proceedings, vol. 3567. CEUR-WS.org, 2023, pp. 33–40. [Online]. Available: http://ceur-ws.org/Vol-3567/paper5.pdf

[12] R. Lämmel and C. Veihoef, "Cracking the 500-Language Problem," *IEEE Software*, vol. 18, no. 6, pp. 78–88, 11 2001. [Online]. Available: https://doi.org/10.1109/52.965809

[13] C. Admiraal, W. van den Brink, M. Gerhold, V. Zaytsev, and C. Zubcu, "Deriving Modernity Signatures of Codebases with Static Analysis," JSS. Available at SSRN: https://doi.org/10.2139/ssrn.4536605, 2024.

[14] S. Yang, T. Kanda, D. Pizzolotto, D. M. Germán, and 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)*. IEEE, 2023, pp. 25–29. [Online]. Available: https://doi.org/10.1109/ICPC58990.2023.00013

[15] C. V. Alexandru, J. J. Merchante, S. Panichella, S. Proksch, H. C. Gall, and G. Robles, "On the Usage of Pythonic Idioms," in *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*. ACM, 2018, pp. 1–11. [Online]. Available: https://doi.org/10.1145/3276954.3276960

[16] P. Leelaprute, B. Chinthanet, S. Wattanakriengkrai, R. G. Kula, P. Jaisri, and T. Ishio, "Does Coding in Pythonic Zen Peak Performance? Preliminary Experiments of Nine Pythonic Idioms at Scale," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension (ICPC)*. ACM, 2022, pp. 575–579. [Online]. Available: https://doi.org/10.1145/3524610.3527879

[17] P. Phan-Udom, N. Wattanakul, T. Sakulniwat, C. Ragkhitwetsagul, T. Sunetnanta, M. Choetkiertikul, and R. G. Kula, "Teddy: Automatic Recommendation of Pythonic Idiom Usage For Pull-Based Software Projects," in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 806–809. [Online]. Available: https://doi.org/10.1109/ICSME46990.2020.00098

[18] Z. Zhang, Z. Xing, X. Xia, X. Xu, and L. Zhu, "Making Python Code Idiomatic by Automatic Refactoring Non-idiomatic Python Code with Pythonic Idioms," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2022, pp. 696–708. [Online]. Available: https://doi.org/10.1145/3540250.3549143

[19] T. Sakulniwat, R. G. Kula, C. Ragkhitwetsagul, M. Choetkiertikul, T. Sunetnanta, D. Wang, T. Ishio, and K. Matsumoto, "Visualizing the Usage of Pythonic Idioms Over Time: A Case Study of the `with open` Idiom," in *Proceedings of he 10th International Workshop on Empirical Software Engineering in Practice (IWESEP)*. IEEE, 2019, pp. 43–48. [Online]. Available: https://doi.org/10.1109/IWESEP49350.2019.00016

[20] A. Farooq and V. Zaytsev, "There is More Than One Way to Zen Your Python," in *Proceedings of the 14th International Conference on Software Language Engineering (SLE)*. ACM, 2021, pp. 68–82. [Online]. Available: https://doi.org/10.1145/3486608.3486909

[21] E. N. Akimova, A. Y. Bersenev, A. A. Deikov, K. S. Kobylkin, A. V. Konygin, I. P. Mezentsev, and 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)*. IEEE, 2021, pp. 141–151. [Online]. Available: https://doi.org/10.1109/APSEC53868.2021.00022

[22] A. Alhefdhi, H. K. Dam, H. Hata, and A. Ghose, "Generating Pseudo-Code from Source Code Using Deep Learning," in *Proceedings of the 25th Australasian Software Engineering Conference (ASWEC)*. IEEE Computer Society, 2018, pp. 21–25. [Online]. Available: https://doi.org/10.1109/ASWEC.2018.00011

[23] R. Sandouka and H. Aljamaan, "Python Code Smells Detection using Conventional Machine Learning Models," *PeerJ Computer Science*, vol. 9, p. e1370, 2023. [Online]. Available: https://doi.org/10.7717/peerj-cs.1370

[24] Z. Chen, L. Chen, W. Ma, and B. Xu, "Detecting Code Smells in Python Programs," in *Proceedings of the International Conference on Software Analysis, Testing and Evolution (SATE)*. IEEE, 2016, pp. 18–23. [Online]. Available: https://doi.org/10.1109/SATE.2016.10

[25] N. Vavrová and V. Zaytsev, "Does Python Smell Like Java?" *The Art, Science and Engineering of Programming (‹Programming›)*, vol. 1, pp. 11–1–11–29, Apr. 2017. [Online]. Available: https://doi.org/10.22152/programming-journal.org/2017/1/11

[26] Z. Ahmed, F. J. Kinjol, and I. J. Ananya, "Comparative Analysis of Six Programming Languages Based on Readability, Writability, and Reliability," in *Proceedings of the 24th International Conference on Computer and Information Technology (ICCIT)*, 2021, pp. 1–6. [Online]. Available: https://doi.org/10.1109/ICCIT54785.2021.9689813

[27] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic Minority Over-sampling Technique," *Journal of Artificial Intelligence Research (JAIR)*, vol. 16, pp. 321–357, 2002. [Online]. Available: https://doi.org/10.1613/JAIR.953

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

[29] A. A. Sawant and P. T. Devanbu, "Naturally!: How Breakthroughs in Natural Language Processing Can Dramatically Help Developers," *IEEE Software*, vol. 38, no. 5, pp. 118–123, 2021. [Online]. Available: https://doi.org/10.1109/MS.2021.3086338

[30] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," in *Empirical Methods in Natural Language Processing (EMNLP)*, ser. Findings of ACL, vol. EMNLP 2020. Association for Computational Linguistics, 2020, pp. 1536–1547. [Online]. Available: https://doi.org/10.18653/V1/2020.FINDINGS-EMNLP.139

[31] C. Latappy, Q. Perez, T. Degueule, J. Falleri, C. Urtado, S. Vauttier, X. Blanc, and C. Teyton, "MLinter: Learning Coding Practices from Examples — Dream or Reality?" in *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2023, pp. 795–804. [Online]. Available: https://doi.org/10.1109/SANER56733.2023.00092

[32] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," https://huggingface.co/microsoft/codebert-base, 2020.

[33] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997. [Online]. Available: https://doi.org/10.1162/NECO.1997.9.8.1735

[34] A. J. Stein, D. Schwartz, Y. Shi, and S. Mancoridis, "Linguistic Approach to Segmenting Source Code," in *Proceedings of the 16th IEEE International Conference on Semantic Computing (ICSC)*. IEEE, 2022, pp. 177–178. [Online]. Available: https://doi.org/10.1109/ICSC52841.2022.00037

[35] X. Wu, W. Zhao, Z. Tan, X. Zhang, and W. Chen, "Research and Implementation of Code Similarity Detection Technology Based on Deep Learning," in *Proceedings of the Ninth IEEE International Conference on Cloud Computing and Intelligent Systems (CCIS)*. IEEE, 2023, pp. 235–239. [Online]. Available: https://doi.org/10.1109/CCIS59572.2023.10262836

[36] Python Software Foundation, "Python Package Index," https://pypi.org, 2003.

[37] T. Oliphant, P. Peterson, E. Jones *et al.*, "SciPy: Fundamental Algorithms for Scientific Computing in Python," https://pypi.org/project/SciPy/, 2001.

[38] M. Grandini, E. Bagli, and G. Visani, "Metrics for Multi-Class Classification: an Overview," *CoRR*, vol. abs/2008.05756, 2020. [Online]. Available: https://arxiv.org/abs/2008.05756

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

[40] P. G. Salgado, "What's New in Python 3.10," https://docs.python.org/3/whatsnew/3.10.html, Oct. 2021.

[41] W. van den Brink, M. Gerhold, and V. Zaytsev, "Deriving Modernity Signatures for PHP Systems with Static Analysis," in *Proceedings of the 22nd IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM NIER)*, 2022, pp. 181–185. [Online]. Available: https://doi.org/10.1109/SCAM55253.2022.00027

[42] T. Bunkerd, D. Wang, R. G. Kula, C. Ragkhitwetsagul, M. Choetkiertikul, T. Sunetnanta, T. Ishio, and K. Matsumoto, "How Do Contributors Impact Code Naturalness? An Exploratory Study of 50 Python Projects," in *Proceedings of the 10th International Workshop on Empirical Software Engineering in Practice (IWESEP)*, 2019, pp. 7–75.