AI-Supported Programming Tasks

Subjects: Computer Science, Artificial Intelligence

Contributor: Sotiris Kotsiantis , Vassilios Verykios , Manolis Tzagarakis

AI-assisted programming or development is defined as the utilization of machine learning models trained on the vast amount of available source code. Its purpose is to support various aspects of programming and, more broadly, software engineering implementation tasks.

AI-assisted programming   code embeddings   transformers

1. Code Summarization

Code summarization involves generating natural language descriptions for source code written in various programming languages, primarily to support documentation generation. During this process, input source code is transformed into a descriptive narrative, typically in English, providing an overview of the code's functionality at the function level.

An enhanced code embedding approach known as Flow2Vec [1] improved the representation of inter-procedural program dependence (value flows) with precision. It accommodated control flows and data flows with alias recognition, mapping them into a low-dimensional vector space. Experiments on 32 open-source projects demonstrated Flow2Vec's effectiveness in enhancing the performance of existing code embedding techniques for code classification and code summarization tasks.

Transformers play a crucial role in generating summaries, involving preprocessing the text by removing unnecessary characters and segmenting them into smaller sentences or phrases. The transformer model, trained on extensive text data, utilizes its attention mechanism to identify key words and phrases, producing a summary based on these essential elements.

Wang et al. [2] introduced Fret, a functional reinforced transformer with BERT, which outperformed existing approaches in both Java and Python. Achieving a BLEU-4 score of 24.32 and a ROUGE-L score of 40.12, Fret demonstrated superior performance in automatic code summarization. For smart contracts, Yang et al. [3] proposed a multi-modal transformer-based code summarization model, showcasing its ability to generate higher-quality code comments compared to state-of-the-art baselines.

Hou et al. [4] presented TreeXFMR, an automatic code summarization paradigm with hierarchical attention, using abstract syntax trees and positional encoding for code representation. Pre-trained and tested on GitHub, TreeXFMR achieved significantly better results than baseline methods.

GypSum [5] incorporated a graph attention network and a pre-trained programming and natural language model for code summarization. Utilizing a dual-copy mechanism, GypSum achieved effective hybrid representations and improved the summary generation process. Gu et al. [6] introduced AdaMo, a method for automated code summarization leveraging adaptive strategies like pre-training and intermediate fine-tuning to optimize latent representations.

Ma et al. [7] proposed a multi-modal fine-grained feature fusion model for code summarization, effectively aligning and fusing information from token and abstract syntax tree modalities. Outperforming current state-of-the-art models, this approach demonstrated superior results.

Gong et al. [8] presented SCRIPT, a structural relative position-guided transformer, using ASTs to capture source code structural dependencies. SCRIPT outperformed existing models on benchmark datasets in terms of BLEU, ROUGE-L, and METEOR metrics. Gao and Lyu [9] proposed M2TS, an AST-based source code summarization technique integrating AST and token features to capture the structure and semantics of source code, demonstrating performance on Java and Python language datasets.

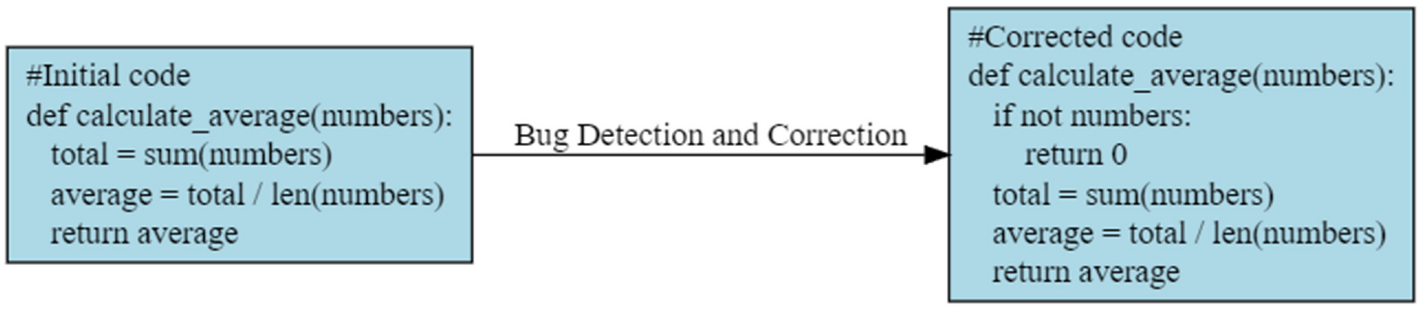
Ferretti and Saletta [10] introduced a novel summarization approach using a pseudo-language to enhance the BRIO model, outperforming CodeBERT and PLBART. The study explored the limitations of existing NLP-based approaches and suggested further research directions.

Choi et al. [11] presented READSUM, a model combining abstractive and extractive approaches for generating concise and informative code summaries. READSUM considered both structural and temporal aspects of input code, utilizing a multi-head self-attention mechanism to create augmented code representations. The extractive procedure verified the relevancy of important keywords, while the abstractive approach generated high-quality summaries considering both structural and temporal information from the source code.

In summary, code embeddings and transformers both play crucial roles in code summarization, yet they operate in distinct ways. Code embeddings typically involve representing code snippets as fixed-length vectors in a continuous vector space, capturing semantic and syntactic information. This approach offers simplicity and efficiency in handling code representations but may struggle with capturing long-range dependencies. On the other hand, transformers excel in modeling sequential data by processing the entire input sequence simultaneously through self-attention mechanisms. This allows them to capture intricate dependencies across code snippets effectively, resulting in more comprehensive summarizations. However, transformers often require larger computational resources compared to code embeddings. Thus, while code embeddings offer efficiency and simplicity, transformers provide a more powerful and context-aware solution for code summarization tasks.

2. Bug Detection and Correction

This task focuses on identifying errors in code (**Figure 1**), emphasizing the detection of unknown errors to enhance software reliability. Traditional bug detection methods rely on manual code reviews, which are often tedious and time-consuming. In contrast, code embedding presents an efficient approach, capable of processing large volumes of code and identifying potential bugs within minutes. The effectiveness of code embedding depends on a diverse training dataset, as a lack of diversity may hinder its ability to capture all types of bugs.



```
#Initial code
def calculate_average(numbers):
  total = sum(numbers)
  average = total / len(numbers)
  return average
```

```
#Corrected code
def calculate_average(numbers):
  if not numbers:
    return 0
  total = sum(numbers)
  average = total / len(numbers)
  return average
```

Figure 1. Code bug detection and correction example.

Aladics et al. [12] demonstrated that representing source code as vectors, based on an abstract syntax tree and the Doc2Vec algorithm, improved bug prediction accuracy and was suitable for machine learning tasks involving source code. Cheng et al. [13] proposed a self-supervised contrastive learning approach for static vulnerability detection, leveraging pre-trained path embedding models to reduce the need for labeled data. Their approach outperformed eight baselines for bug detection in real-world projects.

Hegedus and Ferenc [14] used a machine learning model to filter out false positive code analysis warnings from an open-source Java dataset, achieving an accuracy of 91%, an F1-score of 81.3%, and an AUC of 95.3%. NLP transformers offer an efficient and accurate method for bug detection by analyzing source code, identifying patterns, and detecting inconsistencies indicative of bugs. Bagheri and Hegedus [15] compared text representation methods (word2vec, fastText, and BERT) for detecting vulnerabilities in Python code, with BERT exhibiting the highest accuracy rate (93.8%). Gomes et al. [16] found that BERT-based feature extraction significantly outperformed TF-IDF-based extraction in predicting long-lived bugs, with support vector machines and random forests producing better results when using BERT.

Code summarization, utilizing NLP transformers, presents an approach to bug detection by automatically generating human-readable summaries of code fragments. This method has shown promise in detecting bugs in open-source projects with ample code and bug data available for training.

Evaluation of four new CodeBERT models for predicting software defects demonstrated their ability to improve predictive accuracy across different software versions and projects [17]. The choice of distinct prediction approaches influenced the accuracy of the CodeBERT models.

DistilBERT, a lightweight version of BERT, pre-trained and fine-tuned on various NLP tasks, including bug detection and correction, offers faster and more efficient bug detection, albeit with potentially lower performance than other transformer models. AttSum, a deep attention-based summarization model, surpassed existing models in evaluating bug report titles [18].

Bugsplainer, a transformer-based generative model for explaining software bugs to developers, presented more precise, accurate, concise, and helpful explanations than previous models [19]. Transformers contribute to bug localization, identifying the exact location of bugs in the code. Validation of patches in automated program repair (APR) remains a crucial area, with Csuvik et al. [20] demonstrating the utility of Doc2Vec models in generating patches for JavaScript code.

Mashhadi and Hemmati [21] introduced an automated program repair approach relying on CodeBERT, generating qualitative fixes in various bug cases. Chakraborty et al. [22] created Modit, a multi-modal NMT code editing engine, which outperformed existing models in obtaining correct code patches, especially when developer hints were included.

Generate and validate, a strategy for automatic bug repair using the generative pre-trained transformer (GPT) model, achieved up to 17.25% accuracy [23]. SeqTrans, proposed by Chi et al. [24], demonstrated superior accuracy in addressing certain types of vulnerabilities, outperforming previous strategies in the context of neural machine translation (NMT) technology.

VRepair, an approach by Chen et al. [25], utilized deep learning and transfer learning techniques for automatic software vulnerability repair, showing effectiveness in repairing security vulnerabilities in C. Kim and Yang [26], who utilized the BERT algorithm to predict duplicated bug reports, outperforming existing models and improving bug resolution times.

A technique for developing test oracles, combined with automated testing, improved accuracy by 33%, identifying 57 real-world bugs [27]. da Silva et al. [28] explored various program embeddings and learning models for predictive compilation, with surprisingly simple embeddings performing comparably to more complex ones.

In summary, code embeddings and transformers serve as valuable tools for bug detection and correction, each with its unique strengths. Code embeddings offer a concise representation of code snippets, capturing their semantic and syntactic properties in a fixed-length vector format. This can facilitate efficient similarity comparisons between code segments, aiding in identifying similar bug patterns across projects. However, code embeddings may struggle with capturing complex contextual information and long-range dependencies, potentially leading to limitations in detecting subtle bugs. In contrast, transformers excel in modeling sequential data through self-attention mechanisms, enabling them to capture intricate patterns and contextual information across code segments. This makes transformers particularly effective in detecting and correcting bugs that involve complex interactions and dependencies between code components. Despite the promising results of NLP transformers in bug detection, challenges include the scarcity of large, high-quality datasets and the significant computational resources and training time required. Existing datasets are often language-specific, making generalization to different codebases challenging. Additionally, the resource-intensive nature of NLP transformers may limit their suitability for real-time bug detection.

3. Code Completion

Code completion, a crucial aspect of programming, involves suggesting code to assist programmers in efficiently completing the code they are currently typing. This suggestion can span variable and function names to entire code snippets. The application of transformers in code completion harnesses advanced language models, trained on extensive text data, to enhance developers' coding efficiency. These models exhibit a deep understanding of the context of the code under construction, predicting and suggesting the next code sequence as developers type. This extends beyond basic keyword suggestions, encompassing variable names, function calls, and even the generation of complete code snippets.

The model's proficiency in comprehending syntactic and semantic structures in programming languages ensures accurate and contextually relevant suggestions. It plays a role in identifying and preventing common coding mistakes by offering real-time corrections. Moreover, code completion with transformers often entails providing contextual information such as function signatures, parameter details, and relevant documentation. This not only accelerates the coding process but also aids developers in effectively utilizing various functions and methods.

Roberta [29], another transformer model, has demonstrated impressive results in various natural language processing tasks, showcasing noteworthy performance in code completion. It excels in generating code for diverse programming languages, showcasing a robust understanding of code syntax and context.

Transformer-XL [30], designed to handle longer sequences compared to traditional transformers, has exhibited promising outcomes in code completion tasks, especially when dealing with extensive and intricate sequences. It showcases proficiency in generating code for various programming languages.

CodeFill, proposed by Izadi et al. [31], is a language model for autocompletion leveraging learned structure and naming information. Outperforming several baseline and state-of-the-art models, including GPT-C and TravTrans+, CodeFill excels in both single-token and multi-token prediction. All code and datasets associated with CodeFill are publicly available.

CCMC, presented by Yang and Kuang [32], is a code completion model utilizing a Transformer-XL model for handling long-range dependencies and a pointer network with CopyMask for copying OOV tokens from inputs. The model demonstrates excellent performance in code completion on real-world datasets.

Developers can seamlessly integrate code completion into their preferred integrated development environments (IDEs) or code editors, enhancing the overall coding experience. The interactive and adaptive nature of transformer-based code completion renders it a powerful tool for developers working across various programming languages and frameworks.

Liu et al. [33] introduced a multi-task learning-based pre-trained language model with a transformer-based neural architecture to address challenges in code completion within integrated development environments (IDEs). Experimental results highlight the effectiveness of this approach compared to existing state-of-the-art methods.

BART (bidirectional and auto-regressive transformer), another popular transformer model developed [34], is trained using a combination of supervised and unsupervised learning techniques. Specifically designed for text generation tasks, BART has shown promising results in code generation, achieving state-of-the-art performance in code completion tasks where it predicts the remaining code based on the given context.

A novel neural architecture based on transformer models was proposed and evaluated for autocomplete systems in IDEs, showcasing an accuracy increase of 14–18%. Additionally, an open-source code and data pipeline were released [35]. While transformer models exhibit promise for code completion, further enhancements in accuracy are essential for addressing complex scenarios [36].

In summary, code embeddings and transformers are both valuable tools for code completion, each offering distinct advantages. Code embeddings provide a compact representation of code snippets in a continuous vector space, capturing their semantic and syntactic properties. This allows for efficient retrieval of similar code segments, aiding in suggesting relevant completions based on the context of the code being written. However, code embeddings may struggle with capturing long-range dependencies and contextual nuances, potentially leading to less accurate suggestions in complex coding scenarios. Transformers, on the other hand, excel in modeling sequential data through self-attention mechanisms, enabling them to capture intricate patterns and contextual information across code sequences. This results in more accurate and context-aware code completions, especially in scenarios where understanding broader context and dependencies is crucial.

4. Code Generation Process

Code generation involves the task of creating source code based on constraints specified by the programmer in natural language. Hu et al. [37] introduced a supervised code embedding approach along with a tree representation of code snippets, demonstrating enhanced accuracy and efficiency in generating code from natural language compared to current state-of-the-art methods.

Transformers, a type of neural network architecture widely used for various natural language processing (NLP) tasks, including code generation, utilize an attention mechanism to capture long-term dependencies. They excel in handling sequential data without relying on recurrent connections, making them well-suited for tasks involving code generation.

Transformers can be applied to generate functions or methods based on high-level specifications. Developers can articulate the desired functionality in natural language, and the transformer generates the corresponding code.

Svyatkovskiy et al. [38] introduced IntelliCode Compose, a versatile, multilingual code completion tool capable of predicting arbitrary code tokens and generating correctly structured code lines. It was trained on 1.2 billion lines of code across four languages and utilized in the Visual Studio Code IDE and Azure Notebook.

Gemmell et al. [39] explored Transformer architectures for code generation beyond existing IDE capabilities, proposing a “Relevance Transformer” model. Benchmarking results demonstrated improvement over the current state-of-the-art.

Soliman et al. [40] presented MarianCG-NL-to-Code, a code generation transformer model for generating Python code from natural language descriptions. Outperforming state-of-the-art models, it was downloadable on GitHub and evaluated on CoNaLa and DJANGO datasets.

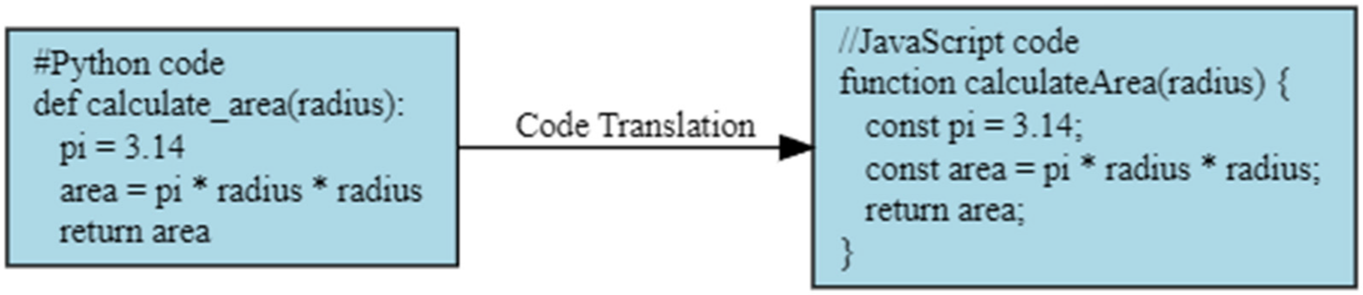
ExploitedGen [41], an exploit code generation approach based on CodeBERT, achieved better accuracy in generating exploit code than existing methods. It incorporated a template-augmented parser and a semantic attention layer, with additional experiments assessing generated code for syntax and semantic accuracy.

Laskari et al. [42] discussed Seq2Code, a transformer-based solution for translating natural language problem statements into Python source code. Using an encoder–decoder transformer design with multi-head attention and separate embeddings for special characters, the model demonstrated improved perplexity compared to similarly structured models.

To summarize the code generation process, code embeddings and transformers offer distinctive approaches, each with its own strengths. Code embeddings condense code snippets into fixed-length vectors, capturing semantic and syntactic information efficiently. This simplifies the generation process by enabling quick retrieval of similar code segments and facilitating straightforward manipulation in vector space. However, code embeddings might struggle with capturing complex dependencies and contextual nuances, potentially limiting their ability to produce diverse and contextually accurate code. In contrast, transformers excel in modeling sequential data through self-attention mechanisms, allowing them to capture intricate patterns and long-range dependencies across code sequences. This enables transformers to generate code with greater context awareness and flexibility, resulting in more accurate and diverse outputs. Nevertheless, transformers typically demand significant computational resources and extensive training data compared to code embeddings.

5. Code Translation

Code translation (**Figure 2**) involves the conversion of source code from one programming language to another, commonly employed for managing legacy source code. Unlike code generation, which takes natural language as input, code translation deals directly with source code. Bui et al. [43] introduced a bilingual neural network (Bi-NN) architecture for automatically classifying Java and C++ programs. Comprising two sub-networks dedicated to Java and C++ source code, Bi-NN utilized an additional neural network layer to recognize similarities in algorithms and data structures across different languages. Evaluation of a code corpus containing 50 diverse algorithms and data structures revealed promising classification results, with increased accuracy attributed to encoding more semantic information from the source code.



```
#Python code
def calculate_area(radius):
  pi = 3.14
  area = pi * radius * radius
  return area
```

```
//JavaScript code
function calculateArea(radius) {
  const pi = 3.14;
  const area = pi * radius * radius;
  return area;
}
```

Figure 2. Code translation example.

In contrast to traditional machine translation methods, transformers, which employ self-attention mechanisms instead of recurrent networks, play a pivotal role in code translation. Transformers facilitate the automatic conversion of source code written in one programming language into its equivalent in another language. This capability proves valuable for tasks such as cross-language code migration, integrating code from different languages, or aiding developers familiar with one language in comprehending and working with code written in another.

Hassan et al. [44] introduced a source code converter based on the neural machine translation transformer model, specializing in converting source code between Java and Swift. The model was trained on a merged dataset, and initial results demonstrated promise in terms of the pipeline and code synthesis procedure.

DeepPseudo, presented by Yang et al. [45], leveraged advancements in sequence-to-sequence learning and code semantic learning to automatically generate pseudo-code from source code. Experiment results indicated DeepPseudo's superiority over seven state-of-the-art models, providing a valuable tool for novice developers to understand programming code more easily.

Alokla et al. [46] proposed a new model for generating pseudocode from source code, achieving higher accuracy compared to previous models. This model utilized similarity measures and deep learning transformer models, demonstrating promising results on two datasets.

DLBT, a deep learning-based transformer model for automatically generating pseudocode from source code [47], tokenized the source code and employed a transformer to assess the relatedness between the source code and its corresponding pseudocode. Tested with Python source code, DLBT achieved accuracy and BLEU scores of 47.32 and 68.49, respectively.

Acharjee et al. [48] suggested a method utilizing natural language processing and a sequence-to-sequence deep learning-based model trained on the SPoC dataset for pseudocode conversion. This approach exhibited increased accuracy and efficiency compared to other techniques, as evaluated using bilingual understudy scoring.

To sum up regarding the realm of code generation translation, both code embeddings and transformers offer distinct advantages. Code embeddings condense code snippets into fixed-length vectors, effectively capturing the semantic and syntactic information essential for translation tasks. This approach simplifies the translation process by enabling quick retrieval of similar code segments and facilitating straightforward manipulation in vector space. However, code embeddings may struggle to capture complex dependencies and nuances present in code, potentially limiting their ability to produce accurate translations. On the other hand, transformers excel in modeling sequential data through self-attention mechanisms, allowing them to capture intricate patterns and long-range dependencies across code sequences. This results in more context-aware translations, with the ability to handle a wide range of coding languages and structures.

6. Code Comment Generation

The objective of this task is the automatic generation of natural language comments for a given code snippet. Shahbazi et al. [49] introduced API2Com, a comment generation model that utilized Application Programming Interface Documentations (API Docs) as external knowledge resources. The authors observed that API Docs could enhance comment generation, especially when there was only one API in the method. However, as the number of APIs increased, the model output was negatively impacted.

ComFormer, proposed by Yang et al. [50], is a novel code comment generator that integrates transformer and fusion method-based hybrid code presentation. Byte-BPE and Sim_SBT were employed to address out-of-vocabulary (OOV) problems during training. The evaluation involved three metrics and a human study comparing ComFormer to seven state-of-the-art baselines from both code comment and neural machine translation (NMT) domains.

Chakraborty et al. [51] introduced a new pre-training objective for language models for source code, aiming to naturalize the code by utilizing its bi-channel structure (formal and informal). The authors employed six categories of semantic maintaining changes to construct unnatural forms of code for model training. After fine-tuning, the model performed on par with CodeT5, exhibiting improved performance for zero-shot and few-shot learning, as well as better comprehension of code features.

Geng et al. [52] proposed a two-stage method for creating natural language comment texts for code. The approach utilized a model interpretation strategy to refine summaries, enhancing accuracy. Thongtanunam et al. [53] developed AutoTransform, an advanced neural machine translation (NMT) model that significantly increased accuracy in automatically transforming code for code review processes. This innovation aimed to reduce developers' time and effort in manual code review.

BASHEXPLAINER [54] automated code comment generation for Bash scripts, outperforming existing methods based on metrics such as BLEU-3/4, METEOR, and ROUGE-L by up to 9.29%, 8.75%, 4.77%, and 3.86%, respectively. Additionally, it offered a browser plug-in to facilitate the understanding of Bash code.

S-Coach, presented by Lin et al. [55], is a two-phase approach to updating software comments. The first phase utilizes a predictive model to determine if comment updates are code-indicative. If affirmative, an off-the-shelf heuristic-based approach is employed; otherwise, a specially-designed deep learning model is leveraged. Results demonstrated that this approach is more effective than the current state-of-the-art by 20%.

In the domain of code comment generation, both code embeddings and transformers play vital roles, each offering distinct advantages. Code embeddings provide a concise representation of code snippets in a continuous vector space, capturing their semantic and syntactic properties. This facilitates the generation of comments by enabling efficient retrieval of similar code segments and assisting in understanding the context for comment generation. However, code embeddings may struggle with capturing the intricacies and nuances of code, potentially leading to less contextually relevant comments. Transformers, on the other hand, excel in modeling sequential data through self-attention mechanisms, allowing them to capture complex patterns and dependencies across code sequences. This results in more context-aware and informative comments that better align with the underlying code logic.

7. Duplicate Code Detection and Similarity

This task involves identifying duplicate code snippets, whether within the same codebase or across different codebases. Transformers play a crucial role in duplicate code detection, automating the identification of redundant or duplicated code segments within a software project. This process is vital for maintaining code quality, enhancing maintainability, and preventing potential issues associated with code redundancy.

Karakatic et al. [56] introduced a novel method for comparing software systems by computing the robust Hausdorff distance between semantic source code embeddings of each program component. The authors utilized a pre-trained neural network model, code2vec, to generate source code vector representations from various open-source libraries. Employing different types of robust Hausdorff distance, the proposed method demonstrated its suitability for gauging semantic similarity.

The presence of code smells and security smells in various training datasets, a fine-tuned transformer-based GPT-Neo model, and a closed-source code generation tool raised concerns about the cautious application of language models to code generation tasks [57].

Yu et al. [58] proposed BEDetector, a two-channel feature extraction method for binary similarity detection, encompassing contextual semantic feature extraction and a neural GAE model. This system achieved impressive detection rates, including 88.8%, 86.7%, and 100% for resilience against CVE vulnerabilities ssl3-get-key-exchange, ssl3-get-new-session-ticket, and udhcp-get-option, respectively.

Mateless et al. [59] developed Pkg2Vec to encode software packages and predict their authors with remarkable accuracy. Comparisons against state-of-the-art algorithms on the ISOT datasets revealed Pkg2Vec’s superior performance, showcasing a 13% increase in accuracy. This demonstrated the efficacy of applying deep learning to improve authorship attribution of software packages, providing deep, interpretable features indicating the unique style and intentions of the programmer.

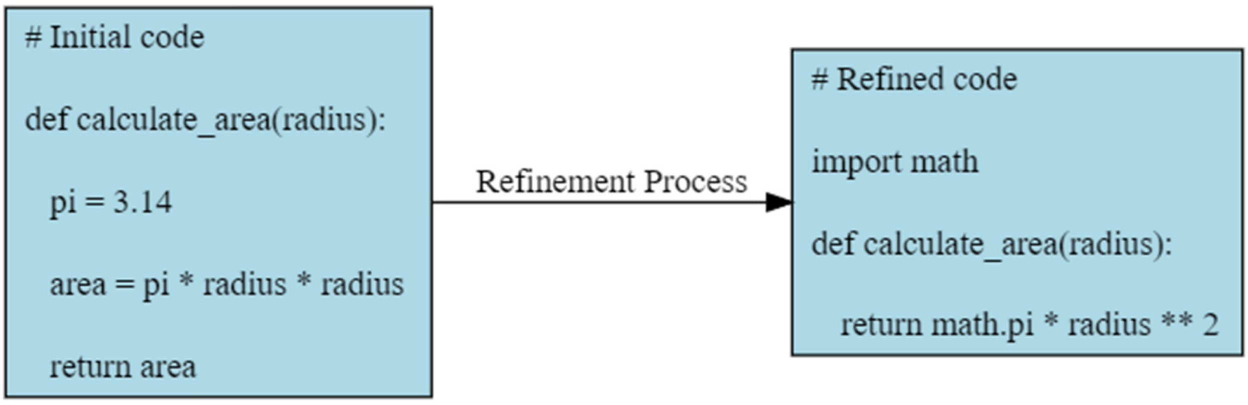
CodeBERT showed effectiveness for Type-1 and Type-4 clone detection, although its performance declined for unseen functionalities. Fine-tuning was identified as a potential avenue to marginally improve recall [60]. Kovacevic et al. [61] investigated the effectiveness of both ML-based and heuristics-based code smell detection models, utilizing different source code representations (metrics and code embeddings) on the large-scale MLCQ dataset. Transfer learning models were evaluated to analyze the impact of mined knowledge on code smell detection.

An efficient transformer-based code clone detection method was proposed by [62], promising accurate and rapid identification of code clones while significantly reducing computational cost.

To sum up, in the realm of duplicate code detection and similarity analysis, both code embeddings and transformers offer unique advantages. Code embeddings distill code snippets into fixed-length vectors, effectively capturing their semantic and syntactic features. This enables efficient comparison and retrieval of similar code segments, facilitating the identification of duplicate code instances. However, code embeddings may struggle to capture complex dependencies and contextual nuances, potentially limiting their effectiveness in detecting subtle similarities. Transformers, on the other hand, excel in modeling sequential data through self-attention mechanisms, allowing them to capture intricate patterns and long-range dependencies across code sequences. This results in more accurate and context-aware similarity analysis, enabling the detection of subtle variations and similarities within code snippets. Nonetheless, transformers typically require larger computational resources and extensive training data compared to code embeddings.

8. Code Refinement

Code refinement (**Figure 3**) involves identifying and correcting pieces of code susceptible to bugs or vulnerabilities. In the work of Liu et al. [63], a software maintenance method was introduced for debugging method names by evaluating the consistency between their names and code to identify discrepancies. Through experiments on over 2.1 million Java methods, the method achieved an F1-measure of 67.9%, surpassing existing techniques by 15%. Notably, the authors successfully fixed 66 inconsistent method names in a live study on projects in the wild.



```
# Initial code

def calculate_area(radius):

  pi = 3.14

  area = pi * radius * radius

  return area
```

```
# Refined code

import math

def calculate_area(radius):

  return math.pi * radius ** 2
```

Figure 3. Code refinement example.

Cabrera Lozoya et al. [64] extended a state-of-the-art approach for representing source code to also include changes in the source code (commits). Transfer learning was then applied to classify security-relevant commits. The study demonstrated that representations based on structural information of the code syntax outperformed token-based representations. Moreover, pre-training with a small dataset (greater than 10^4 samples) for a closely related pretext task showed superior performance compared to pre-training with a larger dataset (more than 10^6 samples) and a loosely related pretext task.

Wang et al. [65] introduced Cognac, a context-guidance method name recommender that incorporated global context from methods related by calls. It utilized prior knowledge to adjust method name recommendations and method name consistency checking tasks. Cognac outperformed existing approaches on four datasets with F-scores of 63.2%, 60.8%, 66.3%, and 68.5%, respectively, achieving an overall accuracy of 76.6%, surpassing MNire by 11.2%, a machine learning approach to check the consistency between the name of a given method and its implementation [66].

Xie et al. [67] proposed DeepLink, a model applying code knowledge graph embeddings and deep learning to identify links between issue reports and code commits for software projects. Evaluation of real-world projects demonstrated its superiority over current state-of-the-art solutions.

Borovits et al. [68] presented an automated procedure using word embeddings and deep learning processes to detect inconsistencies between infrastructure as code (IaC) code units and their names. Experiments on an open-source dataset showed an accuracy range of 78.5% to 91.5% in finding such inconsistencies.

Ma et al. [69] introduced Graph-code2vec, a novel self-supervised pre-training approach using code investigation and graph neural networks to generate agnostic task embeddings for software engineering tasks. The proposed technique proved more effective than existing generic and task-specific learning-based baselines, including GraphCodeBERT.

NaturalCC [70] is an open-source code intelligence toolkit, accessible on the website (http://xcodemind.github.io), built on Fairseq and PyTorch technology. It is designed to enable efficient machine learning-based implementation of code intelligence tasks such as code summarization, code retrieval, and code completion.

In the context of code refinement, both code embeddings and transformers offer distinct advantages. Code embeddings condense code snippets into fixed-length vectors, capturing their semantic and syntactic properties efficiently. This facilitates the refinement process by enabling quick retrieval of similar code segments and aiding in identifying areas for improvement. However, code embeddings may struggle to capture complex dependencies and nuanced coding patterns, potentially limiting their ability to suggest refined solutions accurately. Conversely, transformers excel in modeling sequential data through self-attention mechanisms, enabling them to capture intricate patterns and dependencies across code sequences. This results in more contextually aware refinements, with the ability to suggest solutions that align closely with the underlying logic of the code.

9. Code Security

Code security involves checking source code for exploits that may allow unauthorized access to restricted resources. Zaharia et al. [71] proposed the use of an intermediate representation that strikes a balance between stringency to retain security flaws, as per MITRE standards, and dynamism that does not strictly rely on the lexicon of a programming language. This intermediate representation is based on the semantical clusterization of commands in C/C++ programs through word embeddings. These embeddings are distributed through the formed intermediate representation to different classifiers for recognizing security vulnerability patterns.

In related work, Zaharia et al. [72] developed a security scanning system employing machine learning algorithms to detect various patterns of vulnerabilities listed in the Common Weaknesses Enumeration (CWE) from NIST. This system, independent of the programming language, achieved a recall value exceeding 0.94, providing a robust defense against cyber-attacks.

Barr et al. [73] conducted an in-depth analysis of the Fluoride Bluetooth module’s source code using deep learning, machine learning, heuristics, and combinatorial optimization techniques. They employed byte-pair encoding to lower dimensionality, embedded tokens into a low-dimensional Euclidean space using LSTM, and created a distance matrix based on cosines between vectors of functions. The authors used cluster-editing to segment the graph’s vertices into nearly complete subgraphs, assessing vulnerability risk based on vectors and features of each component.

Saletta and Ferretti [74] discussed a technique using natural language processing to recognize security weaknesses in source code. This involved mapping code to vector space through its abstract syntax trees, and supervised learning to capture distinguishing features among different vulnerabilities. Results demonstrated the model’s ability to accurately recognize various types of security weaknesses.

In the domain of code security, both code embeddings and transformers serve as valuable tools, each with its unique strengths. Code embeddings offer a compact representation of code snippets, capturing their semantic and syntactic properties efficiently. This allows for quick analysis of code similarities, aiding in the identification of potential security vulnerabilities based on patterns observed in known security issues. However, code embeddings may struggle to capture complex interactions and subtle security flaws, potentially leading to limitations in detecting sophisticated attacks. Transformers, on the other hand, excel in modeling sequential data and understanding contextual information through self-attention mechanisms. This enables them to capture intricate patterns and dependencies across code sequences, resulting in a more comprehensive and context-aware analysis of code security. However, transformers typically require larger computational resources and extensive training data compared to code embeddings.

References

1. Sui, Y.; Cheng, X.; Zhang, G.; Wang, H. Flow2Vec: Value-flow-based precise code embedding. Proc. ACM Program. Lang. 2020, 4, 233.
2. Wang, R.; Zhang, H.; Lu, G.; Lyu, L.; Lyu, C. Fret: Functional Reinforced Transformer with BERT for Code Summarization. IEEE Access 2020, 8, 135591–135604.
3. Yang, Z.; Keung, J.; Yu, X.; Gu, X.; Wei, Z.; Ma, X.; Zhang, M. A Multi-Modal Transformer-based Code Summarization Approach for Smart Contracts. In Proceedings of the 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC), Madrid, Spain, 20–21 May 2021.
4. Hou, S.; Chen, L.; Ye, Y. Summarizing Source Code from Structure and Context. In Proceedings of the 2022 International Joint Conference on Neural Networks (IJCNN), Padua, Italy, 18–23 July 2022.
5. Wang, Y.; Dong, Y.; Lu, X.; Zhou, A. GypSum: Learning hybrid representations for code summarization. In Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, Online, 16–17 May 2022.
6. Gu, J.; Salza, P.; Gall, H.C. Assemble Foundation Models for Automatic Code Summarization. In Proceedings of the 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Honolulu, HI, USA, 15–18 March 2022.
7. Ma, Z.; Gao, Y.; Lyu, L.; Lyu, C. MMF3: Neural Code Summarization Based on Multi-Modal Fine-Grained Feature Fusion. In Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Helsinki, Finland, 29–23 September 2022.
8. Gong, Z.; Gao, C.; Wang, Y.; Gu, W.; Peng, Y.; Xu, Z. Source Code Summarization with Structural Relative Position Guided Transformer. In Proceedings of the 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Honolulu, HI, USA, 15–18 March 2022.
9. Gao, Y.; Lyu, C. M2TS: Multi-scale multi-modal approach based on transformer for source code summarization. In Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, Online, 16–17 May 2022.
10. Ferretti, C.; Saletta, M. Naturalness in Source Code Summarization. How Significant is it? In Proceedings of the 2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC), Melbourne, VI, Australia, 15–16 May 2023.
11. Choi, Y.; Na, C.; Kim, H.; Lee, J.-H. READSUM: Retrieval-Augmented Adaptive Transformer for Source Code Summarization. IEEE Access 2023, 11, 51155–51165.
12. Aladics, T.; Jasz, J.; Ferenc, R. Bug Prediction Using Source Code Embedding Based on Doc2Vec. In Computational Science and Its Applications; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2021; pp. 382–397.

13. Cheng, X.; Zhang, G.; Wang, H.; Sui, Y. Path-sensitive code embedding via contrastive learning for software vulnerability detection. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Online, Republic of Korea, 18–22 July 2022.
14. Hegedus, P.; Ferenc, R. Static Code Analysis Alarms Filtering Reloaded: A New Real-World Dataset and its ML-Based Utilization. IEEE Access 2022, 10, 55090–55101.
15. Bagheri, A.; Hegedus, P. A Comparison of Different Source Code Representation Methods for Vulnerability Prediction in Python. In Quality of Information and Communications Technology; Springer: Cham, Switzerland, 2021; pp. 267–281.
16. Gomes, L.; da Silva Torres, R.; Cortes, M.L. BERT- and TF-IDF-based feature extraction for long-lived bug prediction in FLOSS: A comparative study. Inf. Softw. Technol. 2023, 160, 107217.
17. Pan, C.; Lu, M.; Xu, B. An Empirical Study on Software Defect Prediction Using CodeBERT Model. Appl. Sci. 2021, 11, 4793.
18. Ma, X.; Keung, J.W.; Yu, X.; Zou, H.; Zhang, J.; Li, Y. AttSum: A Deep Attention-Based Summarization Model for Bug Report Title Generation. IEEE Trans. Reliab. 2023, 72, 1663–1677.
19. Mahbub, P.; Shuvo, O.; Rahman, M.M. Explaining Software Bugs Leveraging Code Structures in Neural Machine Translation. In Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), Melbourne, VI, Australia, 14–20 May 2023.
20. Csuvik, V.; Horvath, D.; Lajko, M.; Vidacs, L. Exploring Plausible Patches Using Source Code Embeddings in JavaScript. In Proceedings of the 2021 IEEE/ACM International Workshop on Automated Program Repair (APR), Madrid, Spain, 1 June 2021.
21. Mashhadi, E.; Hemmati, H. Applying CodeBERT for Automated Program Repair of Java Simple Bugs. In Proceedings of the 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), Madrid, Spain, 17–19 May 2021.
22. Chakraborty, S.; Ray, B. On Multi-Modal Learning of Editing Source Code. In Proceedings of the 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), Melbourne, VI, Australia, 15–19 November 2021.
23. Lajko, M.; Csuvik, V.; Vidacs, L. Towards JavaScript program repair with generative pre-trained transformer (GPT-2). In Proceedings of the Third International Workshop on Automated Program Repair, Pittsburgh, PA, USA, 19 May 2022.
24. Chi, J.; Qu, Y.; Liu, T.; Zheng, Q.; Yin, H. SeqTrans: Automatic Vulnerability Fix Via Sequence to Sequence Learning. IEEE Trans. Softw. Eng. 2023, 49, 564–585.
25. Chen, Z.; Kommrusch, S.; Monperrus, M. Neural Transfer Learning for Repairing Security Vulnerabilities in C Code. IEEE Trans. Softw. Eng. 2023, 49, 147–165.

26. Kim, T.; Yang, G. Predicting Duplicate in Bug Report Using Topic-Based Duplicate Learning with Fine Tuning-Based BERT Algorithm. IEEE Access 2022, 10, 129666–129675.

27. Dinella, E.; Ryan, G.; Mytkowicz, T.; Lahiri, S.K. TOGA: A neural method for test oracle generation. In Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 21–29 May 2022.

28. da Silva, A.F.; Borin, E.; Pereira, F.M.Q.; Queiroz, N.L.; Napoli, O.O. Program representations for predictive compilation: State of affairs in the early 20's. J. Comput. Lang. 2022, 73, 101171.

29. Liu, Y.; Ott, M.; Goyal, N.; Du, J.; Joshi, M.; Chen, D.; Levy, O.; Lewis, M.; Zettlemoyer, L.; Stoyanov, V. RoBERTa: A Robustly Optimized BERT Pretraining Approach. arXiv 2019, arXiv:1907.11692.

30. Dai, Z.; Yang, Z.; Yang, Y.; Carbonell, J.; Le, Q.V.; Salakhutdinov, R. Transformer-XL: Attentive language models beyond a fixed-length context. In Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics, Florence, Italy, 28 July–2 August 2019; pp. 2978–2988.

31. Izadi, M.; Gismondi, R.; Gousios, G. CodeFill: Multi-token code completion by jointly learning from structure and naming sequences. In Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 21–29 May 2022.

32. Yang, H.; Kuang, L. CCMC: Code Completion with a Memory Mechanism and a Copy Mechanism. In Proceedings of the EASE 2021: Evaluation and Assessment in Software Engineering, Trondheim, Norway, 21–23 June 2021.

33. Liu, F.; Li, G.; Zhao, Y.; Jin, Z. Multi-task learning based pre-trained language model for code completion. In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, Virtual Event Australia, 21–25 December 2020.

34. Lewis, M.; Liu, Y.; Goyal, N.; Ghazvininejad, M.; Mohamed, A.; Levy, O.; Stoyanov, V.; Zettlemoyer, L. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In Proceedings of the Annual Meeting of the Association for Computational Linguistics, Online, 5–10 July 2020; pp. 7871–7880.

35. Kim, S.; Zhao, J.; Tian, Y.; Chandra, S. Code Prediction by Feeding Trees to Transformers. In Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), Madrid, Spania, 22–30 May 2021.

36. Ciniselli, M.; Cooper, N.; Pascarella, L.; Mastropaolo, A.; Aghajani, E.; Poshyvanyk, D.; Di Penta, M.; Bavota, G. An Empirical Study on the Usage of Transformer Models for Code Completion. IEEE Trans. Softw. Eng. 2021, 48, 4818–4837.

37. Hu, H.; Chen, Q.; Liu, Z. Code Generation from Supervised Code Embeddings. In Neural Information Processing; Springer: Cham, Switzerland, 2019; pp. 388–396.

38. Svyatkovskiy, A.; Deng, S.K.; Fu, S.; Sundaresan, N. IntelliCode compose: Code generation using transformer. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Online, 8–13 November 2020.
39. Gemmell, C.; Rossetto, F.; Dalton, J. Relevance Transformer: Generating Concise Code Snippets with Relevance Feedback. In Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval, Virtual Event China, 25–30 July 2020.
40. Soliman, A.S.; Hadhoud, M.M.; Shaheen, S.I. MarianCG: A code generation transformer model inspired by machine translation. J. Eng. Appl. Sci. 2022, 69, 104.
41. Yang, G.; Zhou, Y.; Chen, X.; Zhang, X.; Han, T.; Chen, T. ExploitGen: Template-augmented exploit code generation based on CodeBERT. J. Syst. Softw. 2023, 197, 111577.
42. Laskari, N.K.; Reddy, K.A.N.; Indrasena Reddy, M. Seq2Code: Transformer-Based Encoder-Decoder Model for Python Source Code Generation. In Third Congress on Intelligent Systems; Lecture Notes in Networks and Systems; Springer: Singapore, 2023; pp. 301–309.
43. Bui, N.D.Q.; Yu, Y.; Jiang, L. Bilateral Dependency Neural Networks for Cross-Language Algorithm Classification. In Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Hangzhou, China, 24–27 February 2019.
44. Hassan, M.H.; Mahmoud, O.A.; Mohammed, O.I.; Baraka, A.Y.; Mahmoud, A.T.; Yousef, A.H. Neural Machine Based Mobile Applications Code Translation. In Proceedings of the 2020 2nd Novel Intelligent and Leading Emerging Sciences Conference (NILES), Giza, Egypt, 24–26 October 2020.
45. Yang, G.; Zhou, Y.; Chen, X.; Yu, C. Fine-grained Pseudo-code Generation Method via Code Feature Extraction and Transformer. In Proceedings of the 2021 28th Asia-Pacific Software Engineering Conference (APSEC), Taipei, Taiwan, 6–9 December 2021.
46. Alokla, A.; Gad, W.; Nazih, W.; Aref, M.; Salem, A.-B. Retrieval-Based Transformer Pseudocode Generation. Mathematics 2022, 10, 604.
47. Gad, W.; Alokla, A.; Nazih, W.; Aref, M.; Salem, A. DLBT: Deep Learning-Based Transformer to Generate Pseudo-Code from Source Code. Comput. Mater. Contin. 2022, 70, 3117–3132.
48. Acharjee, U.K.; Arefin, M.; Hossen, K.M.; Uddin, M.N.; Uddin, M.A.; Islam, L. Sequence-to-Sequence Learning-Based Conversion of Pseudo-Code to Source Code Using Neural Translation Approach. IEEE Access 2022, 10, 26730–26742.
49. Shahbazi, R.; Sharma, R.; Fard, F.H. API2Com: On the Improvement of Automatically Generated Code Comments Using API Documentations. In Proceedings of the 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC), Madrid, Spain, 20–21 May 2021.

50. Yang, G.; Chen, X.; Cao, J.; Xu, S.; Cui, Z.; Yu, C.; Liu, K. ComFormer: Code Comment Generation via Transformer and Fusion Method-based Hybrid Code Representation. In Proceedings of the 2021 8th International Conference on Dependable Systems and Their Applications (DSA), Yinchuan, China, 5–6 August 2021.
51. Chakraborty, S.; Ahmed, T.; Ding, Y.; Devanbu, P.T.; Ray, B. NatGen: Generative pre-training by “naturalizing” source code. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Singapore, 14–18 November 2022.
52. Geng, M.; Wang, S.; Dong, D.; Wang, H.; Cao, S.; Zhang, K.; Jin, Z. Interpretation-based Code Summarization. In Proceedings of the 2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC), Melbourne, VI, Australia, 15–16 May 2023.
53. Thongtanunam, P.; Pornprasit, C.; Tantithamthavorn, C. AutoTransform: Automated code transformation to support modern code review process. In Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 21–29 May 2022.
54. Yu, C.; Yang, G.; Chen, X.; Liu, K.; Zhou, Y. BashExplainer: Retrieval-Augmented Bash Code Comment Generation based on Fine-tuned CodeBERT. In Proceeding of the 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME), Limassol, Cyprus, 3–7 October 2022.
55. Lin, B.; Wang, S.; Liu, Z.; Xia, X.; Mao, X. Predictive Comment Updating with Heuristics and AST-Path-Based Neural Learning: A Two-Phase Approach. IEEE Trans. Softw. Eng. 2023, 49, 1640–1660.
56. Karakatic, S.; MiloÅ¡evic, A.; Hericko, T. Software system comparison with semantic source code embeddings. Empir. Softw. Eng. 2022, 27, 70.
57. Siddiq, M.L.; Majumder, S.H.; Mim, M.R.; Jajodia, S.; Santos, J.C.S. An Empirical Study of Code Smells in Transformer-based Code Generation Techniques. In Proceedings of the 2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM), Limassol, Cyprus, 3 October 2022.
58. Yu, L.; Lu, Y.; Shen, Y.; Huang, H.; Zhu, K. BEDetector: A Two-Channel Encoding Method to Detect Vulnerabilities Based on Binary Similarity. IEEE Access 2021, 9, 51631–51645.
59. Mateless, R.; Tsur, O.; Moskovitch, R. Pkg2Vec: Hierarchical package embedding for code authorship attribution. Future Gener. Comput. Syst. 2021, 116, 49–60.
60. Arshad, S.; Abid, S.; Shamail, S. CodeBERT for Code Clone Detection: A Replication Study. In Proceedings of the 2022 IEEE 16th International Workshop on Software Clones (IWSC), Limassol, Cyprus, 2 October 2022.

61. Kovacevic, A.; Slivka, J.; Vidakovic, D.; Grujic, K.-G.; Luburic, N.; Prokic, S.; Sladic, G. Automatic detection of Long Method and God Class code smells through neural source code embeddings. Expert Syst. Appl. 2022, 204, 117607.

62. Zhang, A.; Fang, L.; Ge, C.; Li, P.; Liu, Z. Efficient transformer with code token learner for code clone detection. J. Syst. Softw. 2023, 197, 111557.

63. Liu, K.; Kim, D.; Bissyande, T.F.; Kim, T.; Kim, K.; Koyuncu, A.; Kim, S.; Le Traon, Y. Learning to Spot and Refactor Inconsistent Method Names. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 25–31 May 2019.

64. Cabrera Lozoya, R.; Baumann, A.; Sabetta, A.; Bezzi, M. Commit2Vec: Learning Distributed Representations of Code Changes. SN Comput. Sci. 2021, 2, 150.

65. Wang, S.; Wen, M.; Lin, B.; Mao, X. Lightweight global and local contexts guided method name recommendation with prior knowledge. In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, 23–28 August 2021.

66. Nguyen, S.; Phan, H.; Le, T.; Nguyen, T.N. Suggesting natural method names to check name consistencies. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE ‘20). Association for Computing Machinery, New York, NY, USA; 2020; pp. 1372–1384.

67. Xie, R.; Chen, L.; Ye, W.; Li, Z.; Hu, T.; Du, D.; Zhang, S. DeepLink: A Code Knowledge Graph Based Deep Learning Approach for Issue-Commit Link Recovery. In Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Hangzhou, China, 24–27 February 2019.

68. Borovits, N.; Kumara, I.; Krishnan, P.; Palma, S.D.; Di Nucci, D.; Palomba, F.; Tamburri, D.A.; van den Heuvel, W.-J. DeepIaC: Deep learning-based linguistic anti-pattern detection in IaC. In Proceedings of the 4th ACM SIGSOFT International Workshop on Machine-Learning Techniques for Software-Quality Evaluation, Virtual, USA, 13 November 2020.

69. Ma, W.; Zhao, M.; Soremekun, E.; Hu, Q.; Zhang, J.M.; Papadakis, M.; Cordy, M.; Xie, X.; Traon, Y.L. GraphCode2Vec: Generic code embedding via lexical and program dependence analysis. In Proceedings of the 19th International Conference on Mining Software Repositories, Pittsburg, PA, USA, 23–24 May 2022.

70. Wan, Y.; He, Y.; Bi, Z.; Zhang, J.; Sui, Y.; Zhang, H.; Hashimoto, K.; Jin, H.; Xu, G.; Xiong, C.; et al. NaturalCC: An Open-Source Toolkit for Code Intelligence. In Proceedings of the 2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), Pittsburgh, PA, USA, 22–24 May 2022.

71. Zaharia, S.; Rebedea, T.; Trausan-Matu, S. CWE Pattern Identification using Semantical Clustering of Programming Language Keywords. In Proceedings of the 2021 23rd International Conference on Control Systems and Computer Science (CSCS), Bucharest, Romania, 26–28 May 2021.
72. Zaharia, S.; Rebedea, T.; Trausan-Matu, S. Machine Learning-Based Security Pattern Recognition Techniques for Code Developers. Appl. Sci. 2022, 12, 12463.
73. Barr, J.R.; Shaw, P.; Abu-Khzam, F.N.; Thatcher, T.; Yu, S. Vulnerability Rating of Source Code with Token Embedding and Combinatorial Algorithms. Int. J. Semant. Comput. 2020, 14, 501–516.
74. Saletta, M.; Ferretti, C. A Neural Embedding for Source Code: Security Analysis and CWE Lists. In Proceedings of the 2020 IEEE International Conference on Dependable, Autonomic and Secure Computing, International Conference on Pervasive Intelligence and Computing, International Conference on Cloud and Big Data Computing, International Conference on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech), Calgary, AB, Canada, 17–22 August 2020.