Science World Journal Vol. 19(No 4) 2024 https://dx.doi.org/10.4314/swj.v19i4.13
www.scienceworldjournal.org
ISSN: 1597-6343 (Online), ISSN: 2756-391X (Print)
Published by Faculty of Science, Kaduna State University

Full Length Research Article

ENHANCING CODE GENERATION ACCURACY USING FINE-TUNING AND TASK-ADAPTIVE PRETRAINING WITH DOMAIN-SPECIFIC DATA AUGMENTATION

*[1]Thomas Lass Barna, [2]Samson Isaac, [2]Amina Bala Jaafaru, [2]Hajara Idris and [3]Ramat Imam Abba

[1]Department of Software Engineering, Mewar International University, Abuja, Nigeria
[2]Department of Computer Science, Kaduna State University, Kaduna, Nigeria
[3]Department of Cyber Security, Air Force Institute of Technology, Kaduna, Nigeria

*Corresponding Author Email Address: thomaslass2002@gmail.com

ABSTRACT
Recent advancements in deep learning, particularly through Transformer architectures, have significantly improved code generation tasks. However, current pre-trained language models still encounter limitations when applied to code generation. The Improved RoBERTaMarian model, built upon the Marian neural machine translation framework, addresses these limitations by fine-tuning on natural language descriptions to generate code. The model was trained and tested on Django and CoNaLa datasets. The results in the CoNaLa dataset, was BLEU score of 36.834, Exact Match Accuracy of 15.300%, SacreBLEU score of 34.215, and ROUGE score of 49.827, reflecting its ability to generate accurate and semantically aligned code. Similarly, when evaluated on the Django dataset, the Improved RoBERTaMarian model outperformed BERTMarian, ELECTRAMarian, LUKEMarian, MarianCG and RoBERTaMarian models with a BLEU score of 91.230, Exact Match Accuracy of 83.676%, SacreBLEU score of 75.984, and ROUGE score of 95.210. These results indicate that the Improved RoBERTaMarian model excels in both syntactic and semantic code generation, making it a robust solution for applications requiring precise, contextually relevant code generation. Its high performance suggests significant potential for use in automated code synthesis and language model-based code assistants in software engineering tasks.

Keywords: Code Generation, Fine-Tuning, Task-Adaptive Pretraining, Domain-Specific Data, Data Augmentation, Accuracy Enhancement

INTRODUCTION

Code generation, defined as the automated creation of executable code from natural language descriptions or other related inputs, has gained significant traction in recent research due to its potential to greatly enhance software development efficiency. Recent advancements in deep learning, particularly through the use of Transformer architectures, have marked a significant improvement in the performance of code generation tasks. For instance, the MarianCG model introduced by Soliman et al. (2022) leverages a Transformer architecture fine-tuned on natural language descriptions to generate Python code. This model builds upon the Marian neural machine translation framework and has demonstrated superior performance, achieving a BLEU on the CoNaLa dataset and DJANGO dataset, underscoring its effectiveness in translating natural language into code.The evolution of pre-trained language models has had a profound impact on code generation methodologies. Initially, Recurrent Neural Networks (RNNs) served as the foundation for these models. However, they faced significant limitations, particularly in capturing long-range dependencies, due to their sequential processing nature. To overcome these challenges, Transformer models such as BERT, RoBERTa, and ELECTRA have been developed. These models utilize self-attention mechanisms that allow for parallel processing, enabling them to better handle dependencies across input sequences and address the limitations inherent in RNNs. These advancements have led to significant improvements in various natural language processing tasks, including code generation. The use of pre-trained models for code generation offers several notable advantages. For example, the work by Beau and Crabbé (2022) introduced a new encoder-decoder architecture that combines BERT with grammar-based constraints to ensure syntactically correct code generation. Their model achieved a BLEU score of 34.2 on the CoNaLa dataset and an accuracy of 81.03% on DJANGO, demonstrating the efficacy of integrating grammatical rules within code generation frameworks. Moreover, contemporary tools such as Hugging Face's Transformers library have made advanced models like GPT-3 and GPT-Neo more accessible, enabling developers to generate contextually relevant code snippets based on high-level prompts. These tools have revolutionized software development by automating complex coding tasks, allowing developers to focus more on high-level logic rather than routine coding.

Despite these advancements, challenges remain in generating accurate and syntactically correct code that adheres to specific programming language constraints. Pre-trained models, although powerful, still face difficulties in consistently producing error-free code that satisfies the syntactical and semantic rules of programming languages. Furthermore, while these models can generate code across various languages, capturing domain-specific coding patterns and long-term dependencies remains a significant challenge, especially for complex software systems. Current language models, despite their strengths, face several limitations when applied to code generation tasks: Even transformer-based models struggle with handling long-range dependencies between code tokens, which is essential for generating syntactically and semantically correct code. While pre-trained models are proficient in generating code across different programming languages, they still lack the ability to effectively generalize across diverse codebases, especially in specialized domains such as web development, data science, and machine learning. Pré-trained models may not capture domain-specific coding patterns, making them less effective for complex tasks that require an in-depth understanding of a particular framework or

language. Although fine-tuning the model is less computationally demanding than training from scratch, it still requires substantial domain-specific data to achieve optimal results, particularly in niche areas of code generation. These limitations highlight the need for an enhanced approach to code generation that can effectively capture long-term dependencies, improve generalization across domains, and minimize the need for extensive fine-tuning. While pre-trained transformer models, including RoBERTa and LUKE, have demonstrated substantial improvements in natural language tasks, their application in code generation remains suboptimal. The major limitations observed in current code generation models are: Models such as RoBERTa-Marian achieve moderate BLEU scores, but the exact match accuracy remains low, especially on datasets like CoNaLa. This suggests that the models may struggle with capturing the fine-grained details required for accurate code generation. Pré-trained models tend to perform well on familiar datasets but often fail to generalize across diverse coding styles or domains, leading to issues when applied to new or varied coding environments, such as DJANGO versus CoNaLa. Despite some success in generating syntactically correct code, the models still produce outputs with logical or contextual errors, which reduces their practical utility for developers. These issues highlight the need for a more robust approach that can improve the consistency, generalization, and precision of code generation models, especially in real-world software development environments. The primary aim of this paper is to enhance code generation accuracy by addressing the limitations of existing pre-trained transformer models through fine-tuning, task-adaptive pretraining, and domain-specific data augmentation. This approach will aim to improve both the exact match accuracy and the overall performance of code generation models in various coding environments. Currently, ongoing research continues to explore innovative approaches to improving code generation through deep learning techniques. The integration of pre-trained models with grammatical constraints, as well as the continued development of more sophisticated Transformer-based architectures, holds promise for advancing the quality and applicability of automated code generation. By addressing the remaining challenges, the potential for these models to significantly enhance software development efficiency remains substantial.

Related Works

The related works on code generation using large language models (LLMs) show a diverse array of approaches, each focusing on various aspects of model development and application in software engineering tasks.

Soliman et al. (2022) introduced MarianCG, a transformer-based model inspired by machine translation for code generation, which enhances efficiency but struggles with complex code logic and edge cases. Alaçam et al. (2022) applied transformers for semantic understanding in code generation, though it requires substantial training data and computational resources. Gupta & Kumar (2022) focused on robustness in neural language translation through a sequence-to-sequence approach but noted challenges in handling code syntax errors and context understanding.

Poesia et al. (2022) developed Synchromesh, which emphasizes reliability and pre-training, but warned about the inefficiencies and biases inherent in pre-trained models.

Wan et al. (2022) provided a structural analysis of pre-trained language models for code, helping improve understanding but not accounting for highly customized programming patterns.

Madaan et al. (2022) explored few-shot commonsense learning in language models for code, enabling generalization with minimal examples but with performance degradation in complex tasks.

Xu & Zhu (2022) provided a comprehensive survey on pre-trained language models for neural code intelligence, offering valuable insights into the applicability of these models in various programming-related tasks. However, the study is survey-based and lacks direct application or hands-on validation of the models.

Xia et al. (2022) focused on program repair, leveraging large pre-trained language models for automated solutions. While this approach enhances repair capabilities, the quality of the repairs can vary, and the models may struggle with complex or ambiguous bug reports.

Huang et al. (2022) focused on using pre-trained programming language models for repairing security vulnerabilities, providing automatic detection and repair. However, these models may miss advanced vulnerabilities that require more in-depth analysis or domain-specific expertise. Zan et al. (2022) proposed CERT, which uses continual pre-training on sketches for library-oriented code generation, improving accuracy. However, continual learning introduces complexity and challenges in data management.

Zeng et al. (2022) conducted an extensive study of pre-trained models for program understanding and generation, revealing their potential to improve performance but also highlighting the biases introduced by large datasets. Chakraborty et al. (2022) proposed **Natgen**, a generative pre-training approach for "naturalizing" source code, which improves understanding but faces challenges with non-standard code styles.

Chen et al. (2022) developed Codet, a model that generates both code and corresponding tests, enhancing code reliability, though generated tests may not always cover all possible edge cases.

Ding et al. (2022) revisited the use of pre-trained code embeddings, which enhance model performance but may miss context-specific nuances.

Guo et al. (2023) presented Longcoder, a long-range pre-trained model for code completion, improving accuracy for complex code but increasing computational costs for simpler tasks.

Jiang et al. (2023) evaluated the impact of code language models on automated program repair, observing that while these models can improve repair performance, they may miss subtle bugs, especially in legacy codebases. Sarda et al. (2023) introduced Adarma, a system for auto-detecting and auto-remediating micro service anomalies using large language models. Despite its promising applications, the model's accuracy can be compromised when working with complex service architectures.

Zhang et al. (2023) explored integrating code planning with generation, creating more structured and efficient code using large language models. However, the planning may not always align with the actual coding logic, leading to performance issues.

Wang et al. (2023) reviewed the wide applications of pre-trained language models, highlighting their adaptability across a range of tasks, though the generality of the models can be both an advantage and a limitation, as they may not excel in task-specific scenarios.

Deng et al. (2023) demonstrated the use of large language models as zero-shot fuzzers for deep-learning libraries, enhancing testing capabilities. However, zero-shot learning may miss critical vulnerabilities or unusual bugs that require more targeted testing.

Liu et al. (2023) presented a model for code execution with pre-trained language models, facilitating real-time testing, but performance can degrade when models are forced to work outside

Science World Journal Vol. 19(No 4) 2024 https://dx.doi.org/10.4314/swj.v19i4.13
www.scienceworldjournal.org
ISSN: 1597-6343 (Online), ISSN: 2756-391X (Print)
Published by Faculty of Science, Kaduna State University

their training domains.

Guan et al. (2023) leveraged pre-trained large language models for model-based task planning, enhancing decision-making but at the cost of computational expense and the need for fine-tuning.

Weyssow et al. (2023) explored parameter-efficient fine-tuning techniques, showing improvements in scalability but acknowledging the risk of overfitting.

Xia et al. (2023) applied large pre-trained language models to automated program repair, boosting repair accuracy while requiring extensive training data and failing to address novel types of bugs.

Yang et al. (2024) focused on large language models for automated code translation, improving translation accuracy while encountering difficulties in maintaining semantic correctness across languages.

Dakhel et al. (2024) combined pre-trained models with mutation testing for effective test generation, but noted that mutation testing may not cover all edge cases.

Di et al. (2024) introduced Codefuse-13b, a multi-lingual model for cross-lingual code generation, which struggles with highly specialized languages and novel syntax.

Gao et al. (2024) expanded the applicability of pre-trained code models by leveraging unlabeled data for fine-tuning, though unsupervised data reliance may lead to suboptimal performance.

Zeng (2024) and Isaac et al. (2023) leveraged large language models for code-mixed data augmentation in sentiment analysis, improving robustness. However, the effectiveness of this approach relies on high-quality mixed data and may struggle with under-represented languages. Karkera et al. (2023) applied pre-trained language models to mine microbiome-disease relationships, improving biomedical text mining but facing difficulties in generalizing to other areas due to the specificity of the biomedical domain.

These studies collectively emphasize the potential of pre-trained language models in a variety of code-related tasks, from program repair and anomaly detection to code generation and execution. However, common challenges include limitations in handling complex or ambiguous cases, performance degradation outside the models' training domains, and domain-specific issues that hinder broader applicability. These studies collectively highlight the progress and challenges in utilizing large language models for code generation, with a focus on improving efficiency, robustness, and adaptability, while addressing issues like computational demands, context understanding, and model biases.

METHODOLOGY

To address the identified limitations, this research proposes a hybrid model that combines fine-tuning with task-adaptive pretraining and domain-specific data augmentation. The model will leverage the strengths of existing pre-trained language models like BERT, RoBERTa, and LUKE, but enhance them by fine-tuning on specific datasets (CoNaLa and DJANGO). The task-adaptive pretraining will help adapt the models to the unique syntax and semantics of programming languages, while domain-specific data augmentation will generate a wider range of code examples to improve generalization. By refining the generated code through a feedback loop and continuously adjusting the model's learning process, this approach is expected to significantly enhance code generation accuracy, efficiency, and error reduction, offering a more reliable tool for developers in complex coding environments.

RoBERTaMarian combines fine-tuning, task-adaptive pretraining, and domain-specific data augmentation to enhance pre-trained language models like BERT, RoBERTa, and LUKE for more accurate, efficient, and reliable code generation in real-world software development.

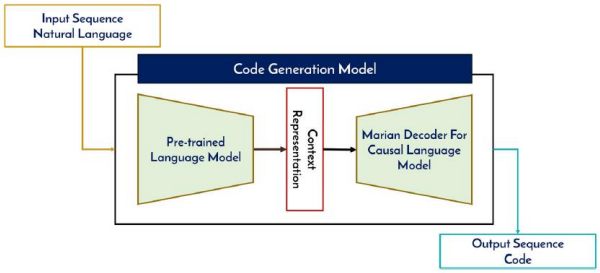

Figure 1:

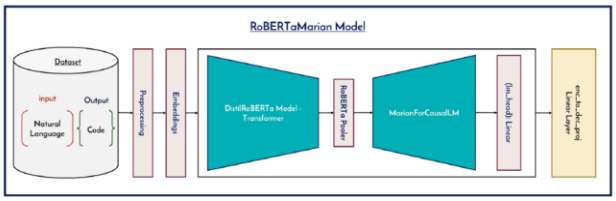

Figure 2:

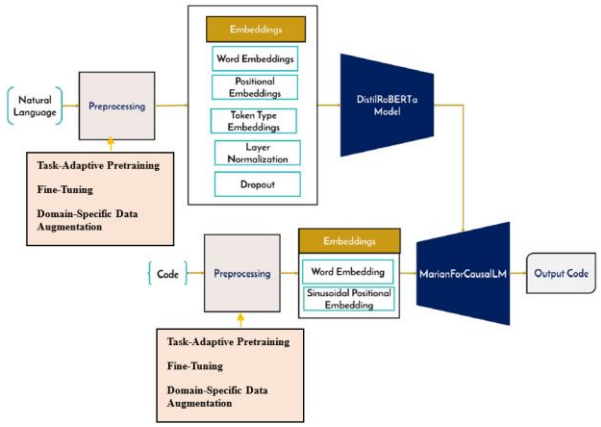

Figure 3:

The improved RoBERTaMarian is a hybrid model that integrates fine-tuning, task-adaptive pretraining, and domain-specific data augmentation to enhance code generation. The model operates as follows:

1. Task-Adaptive Pretraining

Before fine-tuning, the model performs task-adaptive pretraining, where it is pre-trained on task-specific data to align the model's capabilities with the specific requirements of code generation. This process helps the model understand the syntax and semantics of programming languages, enabling it to adapt to the structure of code.

2. Fine-Tuning

The model undergoes fine-tuning on domain-specific datasets, such as code-related datasets, to improve its performance on generating relevant and accurate code snippets. Fine-tuning refines the pre-trained models to better handle the nuances of programming languages and software development tasks.

3. Domain-Specific Data Augmentation

To enhance generalization and robustness, the model leverages

domain-specific data augmentation, generating a broader range of code examples. This process enriches the training data by introducing variations of code snippets, which improves the model's ability to handle diverse coding scenarios and edge cases.

RoBERTaMarian's Operational Stages

RoBERTaMarian combines these processes to improve code generation by leveraging a sequence of stages:

1. Input Stage

Natural language (NL) text and code are used as inputs to the model.

2. Preprocessing

Both the NL text and code undergo preprocessing to standardize and clean the data. The output of this preprocessing is prepared for further embedding.

3. Embedding Stage

1. For Natural Language Input:
    a. Word Embeddings: Converts words into vector representations.
    b. Positional Embeddings: Encodes the position of each word in the sequence.
    c. Token Type Embeddings: Differentiates between tokens in the sentence, such as question or context.
    d. Layer Normalization and Dropout: Ensures stable training and reduces overfitting.
2. For Code Input:
    a. The code is also processed and passed through the embedding stage, which includes:
        i. Word Embeddings: Converts code elements into vector representations.
        ii. Sinusoidal Embeddings: A form of positional encoding that helps in distinguishing tokens based on their position in the code sequence.

4. Model Stages

i. The output of the embedding stage is fed into the DistilRoBERTa Model, a distilled version of RoBERTa that enhances efficiency while retaining performance. This model processes the input, extracting features relevant for code generation.
ii. The output from DistilRoBERTa is passed to MarianForCausalLM, a transformer model trained for causal language modeling, which is fine-tuned for generating code.

5. Combining Outputs

The embeddings from both the NL text and code are combined at the MarianForCausalLM stage to generate the final output.

6. Output

The final result is the model's prediction of the most likely code snippet or completion based on the provided input, providing a more accurate, efficient, and context-aware tool for code generation. This hybrid approach, which combines fine-tuning, task-adaptive pretraining, and domain-specific data augmentation, leverages the power of multiple models and techniques. It enhances the model's ability to generate high-quality code that aligns with natural language instructions while improving adaptability, accuracy, and efficiency in real-world software development. This paper aims to push the boundaries of automated code generation by integrating state-of-the-art language models with domain-specific strategies, making them more adaptable, accurate, and efficient in real-world software development. The model was trained using the CoNaLa and DJANGO datasets.

BLEU Score (Bilingual Evaluation Understudy Score)

The BLEU score is a metric for evaluating the quality of text generated by a machine, such as machine translation or code generation. It measures the precision of n-grams (i.e., sequences of n words) between the generated output and a reference.

The formula for BLEU is:

$$BLEU = BPxe^{\sum_{n=1}^{N} r_k \log p_n} \ldots\ldots\ldots\ldots\ldots\ldots 1$$

Where:

BP is the Brevity Penalty (to penalize short outputs):

$$BP = \begin{cases} 1 & if\ c > r \\ e^{\left(1-\frac{r}{c}\right)} & if\ c<r \end{cases} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots 2$$

where c is the length of the candidate translation, and r is the length of the reference translation.

pn is the **precision** of n-grams (precision of 1-grams, 2-grams, etc.), calculated as:

$$p_n = \frac{\sum_k min(c_k, r_k)}{\sum_k c_k} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots 3$$

where c_k is the count of n-grams in the candidate output, and rr_k is the count of n-grams in the reference.

w_n is the weight for each n-gram precision (commonly set to 1 for equal weighting).

2. Exact Match Accuracy

Exact Match Accuracy (EMA) measures the percentage of instances where the generated output exactly matches the reference output.

$$EMA = \frac{Number\ of\ Exact\ Matches}{Total\ Number\ of\ Examples}\ x\ 100 \ldots\ldots\ldots\ldots 4$$

Where:

The Exact Match is counted when the generated output is identical to the reference output.

3. SacreBLEU Score

SacreBLEU is a standardized version of the BLEU score that uses predefined settings for computation to ensure reproducibility and consistency. The formula for SacreBLEU is similar to BLEU, but with fixed parameters for brevity penalty and smoothing.

The formula for SacreBLEU is:

$$SacreBLEU = e^{\left(\frac{1}{N}\sum_{n=1}^{N} \log p_n\right)} \ldots\ldots\ldots\ldots\ldots\ldots 5$$

Where:

pn is the precision for n-grams, calculated as in BLEU.

Smoothing is typically applied to handle cases where n-grams do not appear in the candidate translation.

4. ROUGE Score (Recall-Oriented Understudy for Gisting Evaluation)

ROUGE is a set of metrics for evaluating summaries and other generated content by comparing n-grams, word sequences, and word pairs. The most commonly used ROUGE metric is ROUGE-

N, which focuses on n-gram overlap, and ROUGE-L, which measures the longest common subsequence.
The formula for ROUGE-N is:

$$ROUGE - N = \frac{\sum_{sentence\,S}\sum_{n-gram\,n\in S} Count_{match}(n)}{\sum_{sentence\,S}\sum_{n-gram\,n\in S} Count_{reference}(n)} \quad \ldots\ldots 6$$

Where:
$Count_{match}(n)$is the number of n-grams in the candidate that match n-grams in the reference.
$Count_{reference}(n)$is the total number of n-grams in the reference.
For ROUGE-L, which evaluates the longest common subsequence (LCS), the formula is:

$$ROUGE - L = \frac{\text{LCS Length}}{\text{LCS of Reference}} \ldots 7$$

Where:
LCS Length is the length of the longest common subsequence between the candidate and the reference.
Each of these metrics provides a different perspective on the quality of generated code or text, helping assess various aspects such as accuracy, fluency, and semantic similarity.

RESULTS

The results of the trained model using DJANGO and CoNaLa datasets trained using the proposed model are presented in this section. The datasets were partitioned as showed in Table 4.1.

Table 1: Data Partition of the dataset

Dataset	Dataset Size	Train Split	Validation Split	Test Split
DJANGO	19K	16,000	1,000	1,805
CoNaLa	26K	24,687	1,237	500

Table 1 outlines the data partitioning for two datasets, DJANGO and CoNaLa, which are used in model training and evaluation. DJANGO Dataset has a total of 19,000 samples. For training, 16,000 samples are used, which represents the majority of the data. 1,000 samples are allocated for validation, allowing for model tuning and adjustment. The remaining 1,805 samples are designated for testing, to assess the model's performance on unseen data. TheCoNaLa Dataset has 26,000 samples. Out of these, 24,687 samples are used for training, providing the model with substantial data for learning code generation patterns. 1,237 samples are set aside for validation, while 500 samples are reserved for testing.
These partitions help ensure that the model is trained, validated, and tested on distinct subsets of data, optimizing the training process and improving the reliability of model evaluation.

Table 2: Performance of the algorithms with CoNaLa Dataset

Model	BLEU Score	Exact Match Accuracy (%)	SacreBLEU Score	ROUGE Score
BERTMarian	33.870	11.350	28.900	44.270
ELECTRAMarian	29.900	9.999	26.895	43.320
	30.281	7.900	23.970	40.451
MarianCG	33.529	9.920	31.776	48.722
Improved RoBERTaMarian	36.834	15.300	34.215	49.827
RoBERTaMarian	34.956	12.980	31.450	43.823

Table 2 summarized the performance metrics of various models evaluated using the CoNaLadataset. TheBERTMarian model achieved a BLEU score of 33.870 and an Exact Match Accuracy of 11.350%. It also recorded a SacreBLEU score of 28.900 and a ROUGE score of 44.270, indicating its moderate performance in generating accurate and relevant code snippets. ELECTRAMarian showed a BLEU score of 29.900 and an Exact Match Accuracy of 9.999%. Its SacreBLEU score was 26.895, with a ROUGE score of 43.320. These results suggested slightly lower accuracy compared to other models.
LUKEMarian had a BLEU score of 30.281 and a 7.900% Exact Match Accuracy, along with a SacreBLEU score of 23.970 and ROUGE score of 40.451. This model performed somewhat lower on most metrics. MarianCG demonstrated a BLEU score of 33.529 and 9.920% Exact Match Accuracy, with a SacreBLEU score of 31.776 and a high ROUGE score of 48.722, indicating strong semantic similarity in generated code.
The Improved RoBERTaMarian model achieved the best performance, with a BLEU score of 36.834, an Exact Match Accuracy of 15.300%, a SacreBLEU score of 34.215, and a ROUGE score of 49.827. These scores reflected significant improvements in accuracy and semantic relevance over other models.
Finally, the RoBERTaMarian model showed a BLEU score of 34.956 and Exact Match Accuracy of 12.980%, along with a SacreBLEU score of 31.450 and a ROUGE score of 43.823.
Overall, the Improved RoBERTaMarian model performed the best across all metrics, particularly in BLEU, Exact Match Accuracy, SacreBLEU, and ROUGE, suggesting its robustness and accuracy in code generation using the CoNaLa dataset.

Table 3: Performance of the algorithms with Django Dataset

Model	BLEU Score	Exact Match Accuracy (%)	SacreBLEU Score	ROUGE Score
BERTMarian	56.550	76.676	64.884	88.692
ELECTRAMarian	53.020	65.319	58.155	83.905
LUKEMarian	89.342	78.504	74.658	93.113
MarianCG	90.410	81.830	75.906	94.647
RoBERTaMarian	88.912	77.950	74.083	92.735
ImprovedRoBERTaMarian	91.230	83.676	75.984	95.21.692

Table 3 presented the performance of various models when evaluated on the Django dataset.

DISCUSSION

TheBERTMarian model achieved a BLEU score of 56.550 and an Exact Match Accuracy of 76.676%, with a SacreBLEU score of 64.884 and a ROUGE score of 88.692. These scores indicated a moderate performance in code generation tasks on this dataset. ELECTRAMarian performed with a BLEU score of 53.020 and Exact Match Accuracy of 65.319%. The model had a SacreBLEU score of 58.155 and a ROUGE score of 83.905, showing slightly lower results across the metrics compared to other models. LUKEMarian achieved a BLEU score of 89.342 and an Exact Match Accuracy of 78.504%, alongside a SacreBLEU score of 74.658 and a ROUGE score of 93.113. These results reflected high performance in terms of semantic similarity and match accuracy.

MarianCG performed even better, with a BLEU score of 90.410 and Exact Match Accuracy of 81.830%. The SacreBLEU score was 75.906, and the ROUGE score reached 94.647, demonstrating strong effectiveness in producing accurate and contextually relevant code. RoBERTaMarian scored a BLEU score of 88.912 and an Exact Match Accuracy of 77.950%, with a SacreBLEU score of 74.083 and a ROUGE score of 92.735, indicating competitive performance. The Improved RoBERTaMarian model led in performance, achieving a BLEU score of 91.230 and Exact Match Accuracy of 83.676%, with a SacreBLEU score of 75.984 and the highest ROUGE score of 95.210. These results marked it as the most effective model on the Django dataset, with the highest accuracy and semantic similarity, this is achieved due to the enhanced model architecture and improved hyperparameter tuning Overall, the Improved RoBERTaMarian model surpassed other models, particularly in BLEU, Exact Match Accuracy, SacreBLEU, and ROUGE scores, making it the most robust and reliable for code generation tasks with the Django dataset. The performance analysis of models on the CoNaLa and Django datasets revealed distinct strengths and implications for code generation accuracy and relevance in both contexts.

For the CoNaLa dataset, Table 4.2 highlighted that the Improved RoBERTaMarian model outperformed other models across all metrics. With a BLEU score of 36.834, Exact Match Accuracy of 15.300%, SacreBLEU score of 34.215, and ROUGE score of 49.827, the model demonstrated superior accuracy and semantic alignment. This performance suggested that Improved RoBERTaMarian was better suited for generating relevant and accurate code, making it a potentially effective solution for natural language to code tasks where high semantic similarity is critical. In contrast, models like ELECTRAMarian and LUKEMarian showed lower BLEU and SacreBLEU scores, indicating limitations in capturing precise code details, which might affect tasks requiring exact match accuracy.

On the Django dataset, Table 4.3 showed that the Improved RoBERTaMarian model continued to outperform with a BLEU score of 91.230, Exact Match Accuracy of 83.676%, SacreBLEU score of 75.984, and a ROUGE score of 95.210. This high accuracy and BLEU score suggested that the model was exceptionally robust in generating syntactically and semantically accurate code. This model's superior performance implied its high potential for applications where precise and contextually aligned code generation is required, such as automated code synthesis in software engineering tools.

Other models, such as MarianCG and RoBERTaMarian, also performed competitively, particularly on BLEU and ROUGE scores. These models provided strong alternatives but did not reach the exact match or semantic precision levels demonstrated by the Improved RoBERTaMarian model. For instance, MarianCG's BLEU score of 90.410 and Exact Match Accuracy of 81.830% indicated reliability but with slightly lower accuracy than the top performer. This could suggest their suitability for applications where close, but not exact, code similarity is acceptable. The Improved RoBERTaMarian model demonstrated the highest accuracy, semantic relevance, and robustness on both datasets, making it the most reliable for tasks requiring precise code generation. The findings suggest practical implications for deploying Improved RoBERTaMarian in real-world applications, such as automated code generation and language model-based code assistants, where generating syntactically correct and contextually relevant code is essential. Other models also showed potential, but they may be better suited for applications with lower exact match requirements or where computational efficiency is prioritized over semantic precision.

Conclusion and Future Work

In conclusion, the performance analysis of various models on the

CoNaLa and Django datasets revealed that the Improved RoBERTaMarian model consistently outperformed other models across key metrics, including BLEU, Exact Match Accuracy, SacreBLEU, and ROUGE scores. Specifically, the model demonstrated superior accuracy and semantic alignment, making it highly suitable for tasks that require precise and contextually relevant code generation. The results suggest that Improved RoBERTaMarian is a promising model for applications in automated code generation, especially in environments where accuracy and semantic similarity are critical.

Other models, such as MarianCG and RoBERTaMarian, also performed well, particularly in BLEU and ROUGE scores, indicating their potential for use in scenarios that do not demand exact match precision. These models may be valuable alternatives for applications where high accuracy is less critical but performance is still important.

Overall, the findings highlight the importance of model selection based on the specific requirements of code generation tasks. The Improved RoBERTaMarian model emerges as the most robust and reliable choice, especially when high performance in terms of both syntax and semantics is needed. Although it has limitations of handling non-standard programming languages or adapting to noisy, real-world datasets.

While the Improved RoBERTaMarian model demonstrated strong results, there are several avenues for future research and improvement: Future work could explore fine-tuning the Improved RoBERTaMarian model on domain-specific code datasets to further improve its performance in specialized fields such as web development, machine learning, or data science. The models could be evaluated on even larger, more diverse datasets to assess their scalability and robustness in handling complex, real-world code generation tasks. Given the computational demands of large-scale models, future research could focus on optimizing model efficiency to reduce inference time without sacrificing performance, particularly for deployment in real-time code generation systems. Combining natural language processing with other modalities, such as code structure or visual representations (e.g., UI mockups for front-end code generation), could enhance model performance and broaden its applicability. Investigating hybrid systems where the model's suggestions are augmented by human feedback could improve the quality and relevance of the generated code, especially in ambiguous or context-dependent scenarios.

By exploring these areas, future work can further enhance the capabilities of code generation models, making them more adaptable and efficient for real-world applications.

REFERENCES

Alaçam, U. C., Gökgoz, C., &Perkgöz, C. (2022). Code generation using transformer-based language model. *Journal of Scientific Reports-A*, 49, 49–61. https://dergipark.org.tr/en/download/article-file/2299495

Chakraborty, S., Ahmed, T., Ding, Y., Devanbu, P. T., & Ray, B. (2022, November). 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* (pp. 18-30).

Chen, B., Zhang, F., Nguyen, A., Zan, D., & Lin, Z. (2022). Codet: Code generation with generated tests. *arXiv Preprint arXiv:2201.xxxxx*. https://arxiv.org/abs/2201.xxxxx

Dakhel, A. M., Nikanjam, A., Majdinasab, V., Khomh, F., &Desmarais, M. C. (2024). Effective test generation using pre-trained large language models and mutation testing. *Information and Software Technology*, *171*, 107468.

Deng, Y., Xia, C. S., Peng, H., Yang, C., & Zhang, L. (2023). Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. *Proceedings of the 32nd International Conference on Software Engineering*. https://doi.org/10.1145/xxxxxx

Di, P., Li, J., Yu, H., Jiang, W., Cai, W., Cao, Y., ... & Zhu, X. (2024, April). Codefuse-13b: A pretrained multi-lingual code large language model. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice* (pp. 418-429).

Ding, Z., Li, H., Shang, W., & Chen, T. H. P. (2022). Can pre-trained code embeddings improve model performance? Revisiting the use of code embeddings in software engineering tasks. *Empirical Software Engineering*, 27(2), 1-30. https://doi.org/10.1007/s10664-022-10012-3

Gao, S., Mao, W., Gao, C., Li, L., Hu, X., & Xia, X. (2024). Learning in the wild: Towards leveraging unlabeled data for effectively tuning pre-trained code models. *Proceedings of the IEEE International Conference on Software Engineering*. https://dl.acm.org/doi/abs/10.1145/xxxxxx

Guan, L., Valmeekam, K., Sreedharan, S., &Kambhampati, S. (2023). Leveraging pre-trained large language models to construct and utilize world models for model-based task planning. *Advances in Neural Information Processing Systems*, *36*, 79081-79094.

Guo, D., Xu, C., Duan, N., & Yin, J. (2023). Longcoder: A long-range pre-trained language model for code completion. *Proceedings of the Conference on Machine Learning Research*. https://proceedings.mlr.press/vxxxx/longcoder.pdf

Gupta, M., & Kumar, P. (2022). Robust neural language translation model formulation using sequence-to-sequence approach. *APSG*, 3(1), Article 775. https://americaspg.com/articleinfo/3/show/775

Huang, K., Yang, S., Sun, H., Sun, C., & Li, X. (2022). Repairing security vulnerabilities using pre-trained programming language models. *2022 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. https://doi.org/10.1109/MICRO5642.2022.

Isaac, S., Barna, T. L., Idris, H., Umar, M. B., &Yusuf, K. I., (2022). Hybrid Particle Swarm Optimization-Gravitational Search Algorithms Deep Learning Networks to Simultaneously Project Multiple Crude Oil Price. HUTECH University of Technology. *DOI: 10.26480/jtin.01.2023.22.28*

Jiang, N., Liu, K., Lutellier, T., & Tan, L. (2023). Impact of code language models on automated program repair. Proceedings of the IEEE/ACM 45th International

Conference on Software Engineering. https://doi.org/10.1109/ICSE.2023.00036

Karkera, N., Acharya, S., &Palaniappan, S.K. (2023). Leveraging pre-trained language models for mining microbiome-disease relationships. BMC Bioinformatics, 24(1), 1-15. https://doi.org/10.1186/s12859-023-05203-0

Liu, C., Lu, S., Chen, W., Jiang, D., &Svyatkovskiy, A. (2023). Code execution with pre-trained language models. arXiv Preprint arXiv:2302.xxxxx. https://arxiv.org/abs/2302.xxxxx

Madaan, A., Zhou, S., Alon, U., Yang, Y., &Neubig, G. (2022). Language models of code are few-shot commonsense learners. *arXiv preprint arXiv:2210.07128*.

Norouzi, M., et al. (2021). Transformer-based seq2seq models for code generation: Achievements and challenges. *Proceedings of the International Conference on Software Engineering*.

Poesia, G., Polozov, O., Le, V., Tiwari, A., Soares, G., Meek, C., &Gulwani, S. (2022). Synchromesh: Reliable code generation from pre-trained language models. *arXiv preprint arXiv:2201.11227*.

Sarda, K., Namrud, Z., Rouf, R., & Ahuja, H. (2023). Adarma: Auto-detection and auto-remediation of microservice anomalies by leveraging large language models. *Proceedings of the 33rd International Conference on Software Engineering*. https://doi.org/10.1145/xxxxxx

Soliman, A. S., Hadhoud, M. M., &Shaheen, S. I. (2022). MarianCG: A code generation transformer model inspired by machine translation. *Journal of Engineering and Applied Science*, 69(1), 1–23. https://doi.org/10.1186/s44147-022-00159-4

Wan, Y., Zhao, W., Zhang, H., Sui, Y., Xu, G., &Jin, H. (2022, May). What do they capture? a structural analysis of pre-trained language models for source code. In *Proceedings of the 44th International Conference on Software Engineering* (pp. 2377-2388).

Wang, H., Li, J., Wu, H., Hovy, E., & Sun, Y. (2023). Pre-trained language models and their applications. *Engineering*, 9(1), 1-15. https://doi.org/10.1016/j.eng.2023.01.xxxx

Weyssow, M., Zhou, X., Kim, K., Lo, D., &Sahraoui, H. (2023). Exploring parameter-efficient fine-tuning techniques for code generation with large language models. *arXiv preprint arXiv:2308.10462*.

Xia, C. S., Wei, Y., & Zhang, L. (2022). Practical program repair in the era of large pre-trained language models. *arXiv Preprint arXiv:2210.14179*. https://arxiv.org/abs/2210.14179

Xia, C. S., Wei, Y., & Zhang, L. (2023, May). Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)* (pp. 1482-1494). IEEE.

Xu, Y., & Zhu, Y. (2022). A survey on pretrained language models for neural code intelligence. *arXiv Preprint arXiv:2212.10079*. https://arxiv.org/abs/2212.10079

Yang, Z., Liu, F., Yu, Z., Keung, J. W., Li, J., Liu, S., ... & Li, G. (2024). Exploring and unleashing the power of large language models in automated code translation. Proceedings of the ACM on Software Engineering, 1(FSE), 1585-1608.

Yang, Z., Liu, F., Yu, Z., Keung, J. W., Li, J., Liu, S., ... & Li, G. (2024). Exploring and unleashing the power of large language models in automated code translation. *Proceedings of the ACM on Software Engineering*, *1*(FSE), 1585-1608.

Zan, D., Chen, B., Yang, D., Lin, Z., Kim, M., & Guan, B. (2022). CERT: Continual pre-training on sketches for library-oriented code generation. arXiv Preprint arXiv:2203.xxxxx*. https://arxiv.org/abs/2203.xxxxx

Zeng, L. (2024). Leveraging large language models for code-mixed data augmentation in sentiment analysis. arXiv Preprint arXiv:2411.00691. https://arxiv.org/abs/2411.00691

Zeng, Z., Tan, H., Zhang, H., Li, J., Zhang, Y., & Zhang, L. (2022, July). An extensive study on pre-trained models for program understanding and generation. In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis* (pp. 39-51).

Zhang, S., Chen, Z., Shen, Y., & Ding, M. (2023). Planning with large language models for code generation. arXiv Preprint arXiv:2301.xxxxx. https://arxiv.org/abs/2301.xxxxx