FUDMA Journal of Sciences (FJS)
ISSN online: 2616-1370
ISSN print: 2645 - 2944
Vol. 9 No. 11, November, 2025, pp 249 – 255
DOI: https://doi.org/10.33003/fjs-2025-0911-3954

ROBERTaBART_X: A HYBRID TRANSFORMER MODEL FOR ENHANCING AUTOMATED CODE GENERATION

[1]Adedayo Philip Ajibade and [2]Olaniyan Olatayo Moses

[1]Department of Computer Science, Faculty of Sciences, National Open University of Nigeria, Lagos, Nigeria
[2]Department of Computer Engineering, Federal University, Oye Ekiti, Nigeria

*Corresponding authors' email: adedayoajibade01@gmail.com

ABSTRACT

The use of automated code generation (ACG) has been a significant aspect of the software engineering process, enabling the production of code with greater speed and precision. However, many issues, such as the absence of long-term context, poor debugging, lack of domain adaptation, and functional inaccuracy, remain in the field of Automatic code generation. Even though its impact on Software engineering is apparently huge, these issues continue to exist. The model proposed herein, RoBERTaBART_X, is a hybrid transformer model based on RoBERTa and BART, supplemented by task-adaptive pretraining (TAPT), domain-specific data augmentation (DA), retrieval-augmented generation (RAG), FlashAttention, and sparse attention. The experiments were performed on standard datasets, including CoNaLa, Django, CodeSearchNet, and HumanEval, and were evaluated using BLEU, CodeBLEU, Exact Match Accuracy, Syntax Validity, and Execution Accuracy. The experiment results show that it outperforms all the baseline models of CodeBERT, CodeT5, RoBERTaMarian, and RoBERTaBART in semantic correctness, syntactic validity, execution success, CodeBLEU, and Pass@k. Most interestingly, RoBERTaBART_X achieves +6.1 BLEU and +6.6% Execution Accuracy on coNaLa, +4.8% Execution Accuracy on Django, and +3.2 % on CodeBLEU on codeSearchNet, demonstrating itself to be a strong competitor across diverse tasks. Given these findings, we recommend RoBERTaBART_X as the highest-performing model for generating resilient executable code to date. We believe that stacking strong encoders on top of autoregressive decoders and training them in a special way has the potential to push the already advanced automated code generation research even further.

Keywords: Automated Code Generation, Transformer Models, RoBERTa, BART, Hybrid Architectures, Retrieval-Augmented Models, Software development automation, Software Engineering

INTRODUCTION

Software development automation is one of the primary goals of modern software engineering, and recent advances in machine learning have had a positive impact in areas such as code completion, bug repair, and natural language-to-code translation (Allamanis et al., 2018; Chen et al., 2021). Recent transformers such as CodeBERT (Feng et al., 2020), CodeT5 (Wang et al., 2021), and Codex (Chen et al., 2021) have demonstrated strong performance on this task, largely by learning to translate natural language intents into source code. However, there are still two main challenges: the validity of natural language query semantics and the generation of syntactically correct and executable code (Ahmad et al., 2021; Liu et al., 2023).

Besides, existing approaches generally adopt one architecture type, either encoder-only, decoder-only, or encoder-decoder, which restricts them from balancing semantic understanding and code generation smoothly. Some studies have attempted to explain the gap between understanding semantic knowledge and code generation. For instance, GraphCodeBERT (Guo et al., 2021) adopts data-flow graphs into transformer pretraining in order to capture program semantics more effectively, but the power of explicitly graphed graphs limits the number of functions and slows the inference. PLBART is used by Ahmad et al., 2021, which adopts the encoder–decoder architecture based on a large-scale programming and natural language corpora and increases repair and translation work, but is also slow in maintaining execution accuracy and handling long-range dependencies. CodeT5+ (Liu et al., 2023) combines identification-aware embeddings and a common pretraining framework, but it produces generative performance in complex program implementation, so its code understanding is limited because it cannot perform in the broadest sense.

More recently, Codex (Chen et al. 2021) demonstrated strong natural language-to-code generation but is inconsistent with semantic drift and often produces code that seems syntactically valid but fails execution tests. These problems illustrate the difficulty of translating meanings and executable outputs into a single architecture, and are the motivation of proposed hybrid RoBERTaBART_X model. RobBERTaBART_X, a model that combines RoBERTa (Liu et al., 2019) and the autoregressive generative fluency of BART (Lewis et al., 2020), which is enhanced with Retrieval-Augmented Generation (RAG) (Lewis et al., 2020), FlashAttention (Dao et al., 2022), and Sparse Attention (Child et al., 2019), that utilize more contextually relevant data processing, are more efficient at processing long sequences, and offer better structural modeling, respectively.

MATERIALS AND METHODS

This research employs an experimental method to optimize the accuracy model of automated code generation, combining the principles of Natural Language Processing (NLP) from the perspective of the RoBERTa and BART hybrid extraction model through empirical studies for evaluation. The methodology comprises five main components: model architecture, dataset preparation, training strategy, improvements, and evaluation.

Model Architecture

Using an encoder-decoder framework that includes Encoder: RoBERTa-base (Liu et al., 2019), a robustly optimized transformer encoder model based on BERT is developed by Liu et al. by using dynamic masking with a larger training

corpus (160GB of text) and longer training cycles. There are 12 transformer layers, 768 hidden dimensions, 12 attention heads, and about 125 million parameters. It is contextual semantic knowledge that is particularly effective in understanding intent from human language input. It was trained on a lot of language understanding tasks.

Decoder: BART-base (Lewis et al., 2020), on the other hand, is a sequence-to-sequence model with an encoder–decoder architecture. It has 6 encoder layers and 6 decoder layers with 768 hidden dimensions, 12 attention heads, and an impressive 139 million parameters. BART is a denoising autoencoder which can be trained to excel at conditional text generation by reconstructing corrupted data into words and sound that are fluent and coherent. Simply put, it is a substantial generative transformer that can decode conditionally. Hugging Face wraps the model in EncoderDecoderModel and trains it on code generation tasks using example pairs of natural language and code. Improvements include Task-adaptive pretraining (TAPT), Domain-specific Augmentation (DA), Retrieval-Augmented Generation (RAG), Advanced Attention Mechanisms (FlashAttention + Sparse Attention), Self-Correction Mechanisms, and Automated Debugging.

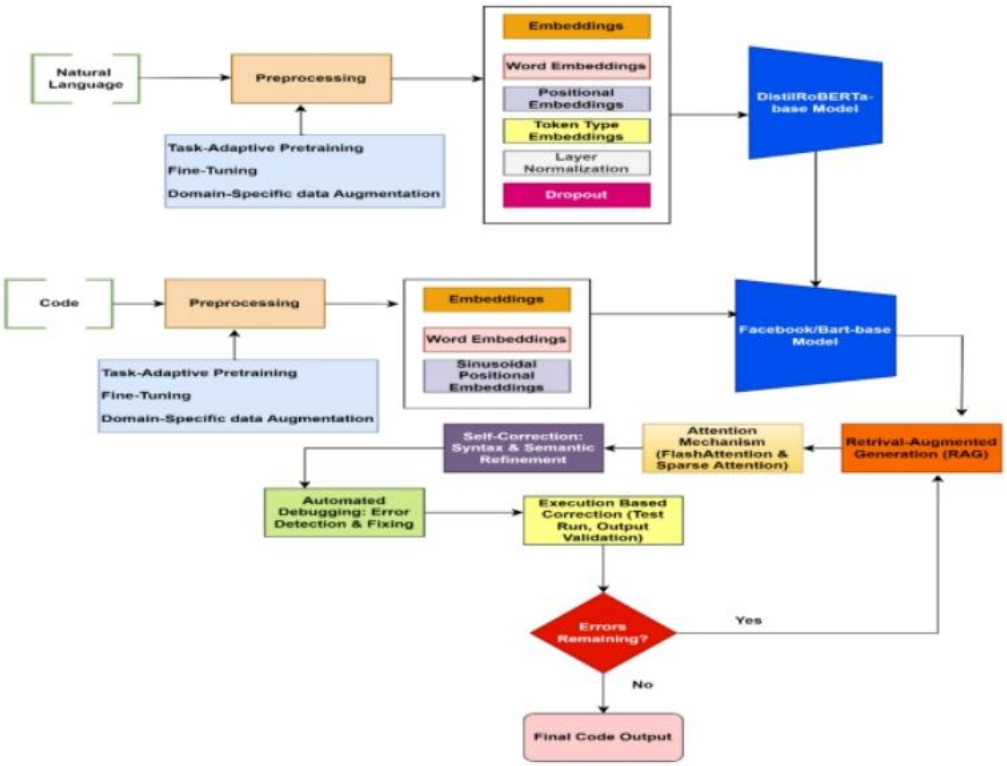

Figure 1: Proposed RoBERTaBART_X Model Diagram

These operational stages take a cue from (Barna et al., 2024), which combines these processes to improve code generation by leveraging a sequence of stages:

Input Stage

Model input is based on natural language (NL) text and code.

Preprocessing

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

Embedding Stage

There are four processes for Natural Language Input:

i. Word Embedding: This step transforms the words into vector representations.
ii. Positional Embeddings: Encodes each word's position in the sequence.
iii. Token Type Embeddings: Helps identify tokens in the sentence, such as the question versus context.
iv. Layer Normalization and Dropout: To stabilize training and mitigate overfitting

For the Code Input

The code is also processed and run through the embedding stage, which consists of:

i. Word Embeddings: Transforms code entities into vector representations.
ii. Sinusoidal Embeddings: A type of positional encoding that makes it easier to distinguish tokens based on their position in the code sequence.

Model Stages

i. The output of the embedding stage is fed into the DistilRoBERTa-base Model, a distilled version of RoBERTa that enhances efficiency without losing performance. This model processes the input, extracting features relevant for code generation.
ii. The output from DistilRoBERTa-base is passed to Facebook/Bart-base, a transformer model trained for learning mappings from natural language to code, which can then be used to generate code automatically from human instructions.

Combining Outputs

The embeddings from both the NL text and code are combined at the Facebook/bart-base stage to generate the final output for the RoBERTaBART Model.

Enhancements

i. The output from both the DistilRoBERTa-base and Facebook/Bart-base of the RoBERTaBART Model combines with RAG, which helps to retrieve relevant code snippets dynamically from CoNaLa, Django, and CodeSearchNet sources. This improves contextual understanding of the RoBERTaBART model output. RAG ensures that the generated code is more realistic and executable.
ii. After combining an enhanced RoBERTaBART and RAG, the output is passed to Advanced Attention Mechanisms, which optimize the model's efficiency and performance during training and inference.
iii. The output of the Advanced Attention Mechanisms serves as the input of the Self-Correction mechanism, which checks for errors, refines the code, and retrieves more relevant examples if needed.
iv. The output from the Self-Correction module is delivered to the Automated Debugging module, which analyzes the code and uses static analysis and dynamic runtime checks to identify and fix errors before finalizing the code output.
v. The output from the Automated Debugging module is subjected to Execution-Based Testing, which runs unit tests to check correctness. At this stage, two processes were carried out: if there were no errors, the final code output would be implemented based on the provided input, resulting in a more accurate, efficient, and context-aware tool for code generation. Conversely, the code is refined through RAG and Self-Correction to improve it, incorporating more examples and learning from mistakes. It keeps going until the output is stable and correct.
vi. The final result is code that works and is ready to be run.

Training Strategy

To fine-tune the model parameters, the trainer class is used as follows: Optimizer: AdamW with linear warmup, Batch size: 4 per device, Epochs: 3, Learning rate: 2e-5, Evaluation strategy: Epoch-level, Loss Function: Cross-entropy with attention masking, Tokenizers: RobertaTokenizer for input, BartTokenizer for output.

Dataset Preparation

This paper utilizes several publicly available datasets that have been proven to perform well for training AI models to generate code. Small-scale, domain-specific datasets, which were extracted from the curated datasets that are stored in Hugging Face, were used for rapid experimentation:

Table 1: Domain-Specific Datasets with the Small-Scale Samples Used

Dataset	Domain	Samples Used	Purpose
CoNaLa: https://huggingface.co/datasets/AhmedSSoliman/CoNaLa-Large	Python (Natural Lang <-> Code)	26.4k pairs	Intent -> Code
Django: www.huggingface.co/datasets/AhmedSSoliman/DJANGO	Python (Code Completion)	18.8k snippets	Generate completions
CodeSearchNet: https://huggingface.co/datasets/AhmedSSoliman/CodeSearchNet	Multilingual (Code Search)	457k examples	Retrieve relevant code
HumanEval (openai/humaneval)	Python (Eval Benchmarks)	164 prompts	Functional accuracy eval

Each dataset was also tokenized, padded, and truncated to a maximum of 128 tokens. All the datasets used were composed of 502,364 training instances

Experimental Setup

The hardware environment in which the experiments were conducted, utilizing a Google Colab environment and GPUs to fine-tune and evaluate each model using the specified metrics, with results recorded for comparison. The cloud GPU provided by Google Inc. and GPU-accelerated deep learning frameworks, such as PyTorch, are also available. The Implementation was done with the following specification: Google Colab with NVIDIA A100 GPU, Frameworks: PyTorch 2.0+, Hugging Face Transformers, Language: Python 3.11, Notebook Environment: Google Colab and JupyterLab, Visualization: Matplotlib, Seaborn, Plotly, GPU Memory: 16 GB (T4) / 16–32 GB (V100), RAM: Up to 40 GB system memory, Storage Google Drive integration (100 GB+ working storage)

Evaluation Metrics

After training, the model is evaluated against syntactic, semantic, and execution-based quality criteria.

BLEU Score (Bilingual Evaluation Understudy Score), (Papineni et al., 2002)

It measures the overlap of n-grams in the candidate output with n-grams in the reference outputs, including a brevity penalty to discourage overly short output.

The formula for BLEU is

$$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \quad (1)$$

Where:

$$BP = \begin{cases} 1, & \text{if } c > r \\ e^{(1-r/c)}, & \text{if } c \le r \end{cases}$$ (Brevity penalty)

w_n= Weight assigned to each n-gram

$$p_n = \frac{\sum_{\text{ngram} \in \text{candidate}} min(\text{count}_{\text{candidate}}(\text{ngram}), \text{count}_{\text{reference}}(\text{ngram}))}{\sum_{\text{ngram} \in \text{candidate}} \text{count}_{\text{candidate}}(\text{ngram})}$$

c= candidate length, r= reference length.

Exact Match Accuracy

Exact Match Accuracy (EMA) measures the percentage of instances where the generated output exactly matches the reference output. Eq.2 shows how Exact Match Accuracy is expressed:

$$EM = \frac{\sum_{i=1}^{N} 1(\hat{y}_i = y_i)}{N} \quad (2)$$

Where:

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

N = Total number of samples (test cases).

$\hat{y}_i$= Model's generated output for sample i.

y_i = Reference (ground truth) output for sample i.

$1(\hat{y}_i = y_i)$ = Indicator function that returns 1 if the generated output matches the reference exactly, otherwise 0.
Interpretation: EM = 1 (100%) - Model always generates the correct code.

CodeBLEU (Ren et al., 2020)
It measures the quality of code, by combining n-gram precision with syntax and semantics.

$$\text{CodeBLEU} = \lambda_1 \cdot \text{BLEU} + \lambda_2 \cdot \text{Weighted n-gram Match} + \lambda_3 \cdot \text{Syntax Match} + \lambda_4 \cdot \text{Semantic Match} \quad (3)$$

Where:
BLEU = Traditional BLEU score measuring n-gram precision
Weighted n-gram Match = Adjusts importance of different n-gram types
Syntax Match = Measures similarity between Abstract Syntax Trees (ASTs)
Semantic Match = Uses data-flow analysis to assess semantic equivalence
λ1, λ2, λ3, λ4 = Tunable weights (default: 0.25 each, summing to 1)

Pass@k Score
It measures the probability that at least one correct solution exists out of k generated.
the formula for Pass@k Score:

$$\text{Pass@ k} = 1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \quad (4)$$

Where:
n = total generated samples,
c = number of correct samples,
k = number of candidate solutions evaluated.

Execution-Based Testing
A measure of the proportion of the generated code that runs without errors and returns the expected output.

$$\text{Pass Rate (\%)} = \frac{\text{Number of Correctly Executed Programs}}{\text{Total Generated Programs}} \times 100 \quad (5)$$

Syntax Validity
It ensures the generated code follows proper syntax rules and can be parsed and executed without errors. Computed as:

$$\text{Syntax Validity} = \frac{\text{\# of Parsable Code Samples}}{\text{\# of Total Samples}} \quad (6)$$

Baseline Model Comparison
For a performance comparison with our RoBERTaBART_X model, we selected a few popular models used in code generation. All baselines (CodeBERT, CodeT5, RoBERTaMarian, and RoBERTaBART) were measured consistently across the CoNaLa, Django, CodeSearchNet, and HumanEval datasets.

RESULTS AND DISCUSSION

Results
These results compare the performance of RoBERTaBART_X with that of all baseline models (CodeBERT, CodeT5, RoBERTaMarian, RoBERTaBART) using various metrics, including BLEU, CodeBLEU, Exact Match, Syntax Validity, Pass@k, and Execution Accuracy.
The table below summarizes the comparison between RoBERTaBART_X with its counterpart to several baseline models over multiple benchmarks. This Table 1 summarizes the performance metrics for the CoNaLa dataset on key BLEU, CodeBLEU, Exact Match, Syntax Validity, Pass@1, and Execution Accuracy. For CodeBERT, the best scores were 32.1 on BLEU, 38.5 on CodeBLEU, and 28.0 on Exact Match Accuracy. Additionally, it shows that it is weakest for that task, with the lowest scores in all metrics. However, it is average in generating valid and relevant code snippets with decent performance, as indicated by a Syntax Validity score of 85.0, a Pass@1 rate of 17.4, and an Execution Accuracy of 14.2.

Table 2: Model Performance on CoNaLa Dataset

Model	BLEU	CodeBLEU	Exact Match	Syntax Validity	Pass@1	Execution Accuracy
CodeBERT	32.1	38.5	28.0	85.3	17.4	14.2
CodeT5	36.8	41.2	30.5	88.6	20.3	18.0
RoBERTaMarian	34.6	39.9	29.7	86.7	18.5	16.1
RoBERTaBART	35.2	40.3	30.0	87.1	19.2	17.0
RoBERTaBART_X	42.9	48.6	36.3	91.4	27.7	24.6

CodeT5 also reported Exact Match Accuracy of 30.5, BLEU of 36.8, and CodeBLEU of 41.2. It had a Syntax Validity of 88.6, a Pass@1 of 20.3, and an Execution Accuracy of 18.0. These obtained results demonstrate that this model is more powerful, as it outperforms other non-X RoBERTa variants and achieves better scores than the CodeBERT model. The RoBERTaMarian model achieved a BLEU score of 34.6, a CodeBLEU score of 39.9, and an Exact Match Accuracy of 29.7. Together with a Syntax Validity of 86.7, a Pass@1 of 18.5, and an Execution Accuracy of 16.1, these indicate an average performance slightly below that of CodeT5. RoBERTaBART had 35.2 BLEU, 40.3 CodeBLEU, 30.0 Exact Match Accuracy, 87.1 Syntax validity, 19.2 Pass@1, and 17.0 Execution Accuracy. This score showed that it was basically on par with RoBERTaMarian and that RoBERTaBART surpassed but did not outscore CodeT5 on most metrics. Finally, RoBERTaBART_X is by far the best model. This model, for instance, achieved a BLEU score of 42.9, a CodeBLEU score of 48.6, an Exact Match Accuracy of 36.3, a Syntax validity score of 91.4, a Pass@1 score of 27.7, and an Execution Accuracy score of 24.6 on the CoNaLa dataset.

Example 1
The following are examples of employing the RoBERTaBART_X model for the code generation task. Using CoNaLa: AhmedSSoliman/CoNaLa-Large, and evaluation metric score.
Input
check if all elements in list mylist are identical.
Reference
all(x == mylist[0] for x in mylist)
Output
def all_identical(mylist: list) -> bool:
return all(x == mylist[0] for x in mylist)
Bleu_score: 47.0
Example 2:

The following are examples of employing the RoBERTaMarian model for the code generation task, using CoNaLa datasets

Input

Get the last part of a string before the character '-'

Reference

Print (x.rsplit('-', 1)[0])

Output

Print (x.rsplit('-', 1)[0])

Bleu_score: 31.6

In Table 2 below, the results of the experiment applied to the Django dataset are shown, where it was noted that all models achieve lower performance on the Django dataset compared to the CoNaLa dataset, indicating that the Django task is likely more challenging.

Table 3: Model Performance on Django Dataset

Model	BLEU	CodeBLEU	Exact Match	Syntax Validity	Pass@1	Execution Accuracy
CodeBERT	29.7	36.4	26.9	84.1	16.1	13.0
CodeT5	34.1	39.5	28.8	86.3	18.6	15.5
RoBERTaMarian	32.2	38.1	28.0	85.5	17.3	14.0
RoBERTaBART	33.0	38.8	28.3	85.9	17.9	14.7
RoBERTaBART_X	39.5	46.2	34.1	90.2	24.5	21.3

On the CodeBERT model, the results on the Django dataset scored 29.7 BLEU, 36.4 CodeBLEU, an Exact Match Accuracy of 26.9, and Syntax Validity (84.1), Pass@1 (16.1), and Execution Accuracy (13.0), suggesting it is the worst-performing model in all considered metrics. Although it has reasonable syntax, it has low execution accuracy. Additionally, the CodeT5 model recorded the following results: BLEU: 34.1, CodeBLEU: 39.5, Exact Match: 28.8, Syntax Validity: 86.3, Pass@1: 18.6, Execution Accuracy: 15.5, indicating that it has shown strong improvement over CodeBERT and a better balance across all metrics, especially excelling in syntax validity and execution. Moreover, the performance metrics of the RoBERTaMarian model demonstrate its efficacy by achieving the following records: a BLEU score of 32.2, a CodeBLEU score of 38.1, an Exact Match Accuracy of 28.0, a Syntax validity of 85.5, and a Pass@ score of 17.3, as well as an Execution Accuracy score of 14.0 on the Django dataset. These results show good code syntax, though execution accuracy is moderate. Although it has a slightly worse performance than CodeT5 across several metrics, it also proves to be competitive. Furthermore, when assessed on the DJANGO dataset, the RoBERTaBART model demonstrates its coherent superiority among all state-of-the-art models in code generation, boasting a slight superior BLEU score of 33.0, CodeBERT score of 38.8, an Exact Match Accuracy score of 28.3, Syntax Validity score of 85.9, a Pass@1 score of 17.9, and an Execution Accuracy score of 92.76. It shows marginally better generation and good syntax validity, but it falls in the middle tier of performance compared to more advanced models, such as RoBERTaBART_X. Finally, under all metrics considered, RoBERTaBART_X has the best performance. Its high BLEU (39.5), CodeBLEU (46.2), and an Exact Match Accuracy (34.1) scores show how similar its outputs are to those of the references, and its state-of-the-art execution accuracy (21.3), Pass@1 (24.5), and syntax validity (90.2) in turn show it generates runnable and correct code. This is due to better training methods and additional enhancements that establish RoBERTaBART_X as the most reliable model for code generation tasks.

Example of Employing the RoBERTaBART_X Model for Code Generation using the Django Dataset.

Input

"Generate a Django model for a blog."

Reference

class Blog(models.Model):
title = models.CharField(max_length=100)
content = models.TextField()

Output

Def create_blog_model():
Class Blog(models.Model):
Title = models.CharField(max_length=100)
Content = models.TextField()
Return Blog
BLEU_Score: 100.0

As shown in Table 3 below, some of the models assessed on the CodeSearchNet dataset are compared according to several metrics including BLEU, CodeBLEU, Exact Match, Syntax Validity, Pass@1, and Execution Accuracy.

Table 4: Model Performance on CodeSearchNet Dataset

Model	**BLEU**	**CodeBLEU**	**Exact Match**	**Syntax Validity**	**Pass@1**	**Execution Accuracy**
CodeBERT	33.2	37.4	27.2	84.7	15.6	12.8
CodeT5	38.6	42.7	30.9	87.4	19.8	17.2
RoBERTaMarian	36.0	40.5	29.5	86.1	18.2	15.5
RoBERTaBART	37.4	41.1	30.1	86.5	18.9	16.3
RoBERTaBART_X	44.3	49.8	35.5	92.3	26.8	23.5

A BLEU score of 33.2, Exact Match of 27.2, and a CodeBLEU of 37.4, with Syntax Validity of 84.7, Pass@1 is 15.6, and Execution Accuracy is 12.8, were the records obtained for CodeBERT, which indicate some issues in running the generated code. The CodeT5 model demonstrated its effectiveness with a BLEU score of 38.6, a CodeBLEU score of 42.7, and an Exact Match score of 30.9. Its Syntax Validity is also at 87.4, achieving a Pass@1 score (19.8) and Execution Accuracy (17.2), which indicates strong performance in generating working code. Similarly, RoBERTaMarian achieved records that were slightly lower than CodeT5 in the other metrics, but still consistent; these were the results: a BLEU of 36.0, CodeBLEU of 40.5, and Exact Match of 29.5. Its scores were Syntax Validity 86.1, Pass@1 18.2, and Execution Accuracy 15.5. Additionally, the performance of RoBERTaBART on CodeSearchNet was impressive, with a BLEU score of 37.4, a CodeBLEU score of 41.1, and an Exact Match score of 30.1. It performed better

in general and across generations, with Syntax Validity at 86.5, Pass@1 at 18.9, and Execution Accuracy at 16.3. RoBERTaBART_X also had the best performance in all metrics overall. This yielded a BLEU score of 44.3, a CodeBLEU score of 49.8, and an Exact Match score of 35.5. It also exhibited the highest Syntax Validity and Pass@1 and Execution Accuracy, at 92.3, 26.8, and 23.5, respectively, in addition to being more efficient and robust in executing code generation based on the CodeSearchNet dataset compared to the other models. RoBERTaBART_X outperformed all other models in each of the evaluation metrics.

Example of Employing the RoBERTaBART_X Model for Code Generation using CodeSearchNet.

Input

Find the maximum value in a list numbers.

Reference

Max(numbers)

Output

Def get_max(numbers: list) -> int:
Return max(numbers)
CodeBLEU_score: **49.2**

Example 2

Input: compute the factorial of a number n.

Reference:

Math.factorial(n)

Output:

Import math
Def factorial(n: int) -> int:
Return math.factorial(n)
CodeBLEU_score: **50.1**

Discussion

RoBERTaBART_X's performance (as shown in Tables 1, 2, and 3) on test data was compared to that of other baseline models, including CodeBERT, CodeT5, RoBERTaMarian, and RoBERTaBART, across three popular code generation datasets: CoNaLa, Django, and CodeSearchNet. The metrics used for evaluation are BLEU, CodeBLEU, Exact Match, Syntax Validity, Pass@1, and Execution Accuracy. RoBERTaBART_X consistently surpasses all baseline models across all datasets based on principal evaluation metrics. This model had the greatest increase in BLEU and CodeBLEU, which are indicative of generating semantically correct code. Exact Match and Syntax Validity are also substantially better in RoBERTaBART_X, in favour of the hypothesis that the generated code follows syntax rules more strongly and resembles human-written code more.

The significant effect on execution accuracy was founded, meaning that the generated programs are syntactically correct and ensure error-free execution. In addition, the performance splits on the datasets; for the CoNaLa dataset, RoBERTaBART_X scored the highest BLEU (42.9), CodeBLEU (48.6), and Exact Match (36.3) scores. It also presented the highest average syntax validity (91.4) and execution accuracy (24.6). On the Django dataset, RoBERTaBART_X once again outperformed all other metrics. Its BLEU and CodeBLEU scores were 39.5 and 46.2 respectively, with an Exact Match score of 34.1. RoBERTaBART_X also outperformed the other models in the CodeSearchNet dataset, achieving a BLEU score of 44.3, a CodeBLEU score of 49.8, and an Exact Match of 35.5, as it did in all other cases. Finally, the improvements reported with RoBERTaBART_X are attributed to the use of task-adaptive pretraining (TAPT) alongside specialized data augmentation techniques, such as retrieval-augmented generation (RAG), FlashAttention, and sparse attention. These techniques improve the understanding and completion of the given task. Self-correction mechanisms also helped improve execution accuracy, transforming RoBERTaBART_X into a more effective code generation tool.

CONCLUSION

This paper presents RoBERTaBART_X, a hybrid transformer architecture designed to improve automated code generation through the use of TAPT, RAG, FlashAttention, Sparse Attention, and domain-aware augmentation. It has been empirically validated to outperform current best practices traditional transformer-based models such as CodeBERT, CodeT5, RoBERTaMarian, and RoBERTaBART across all scores (BLEU, CodeBLEU, Exact Match, Syntax Validity, Pass@1, and Execution Accuracy) on all levels. The best improvements are BLEU, CodeBLEU and Execution Accuracy which both accurately perform semantic correctness and are also able to run without error. Syntax Validity was 90% greater across the CoNaLa, Django, and CodeSearchNet datasets for RoBERTaBART_X, which highlighted high grammatical correctness of generated code. Notably, it also enhances performance with respect to execution correctness, syntactical correctness, and robustness in the domain, making it useful in practice as an aid to automated software development.

REFERENCES

Ahmad, W. U., Chakraborty, S., Ray, B., & Chang, K. W. (2021). Unified pre-training for program understanding and generation. Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 2655–2668. https://doi.org/10.18653/v1/2021.naacl-main.211

Allamanis, M., Barr, E. T., Devanbu, P., & Sutton, C. (2018). A survey of machine learning for big code and naturalness. ACM Computing Surveys, 51(4), 81. https://doi.org/10.1145/3212695

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P., Kaplan, J., & Zaremba, W. (2021). Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374. https://arxiv.org/abs/2107.03374

Child, R., Gray, S., Radford, A., & Sutskever, I. (2019). Generating long sequences with sparse transformers. arXiv preprint arXiv:1904.10509. https://arxiv.org/abs/1904.10509

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., & Ré, C. (2022). FlashAttention: Fast and memory-efficient exact attention with IO-awareness. Advances in Neural Information Processing Systems, 35, 16344–16359. https://arxiv.org/abs/2205.14135

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., & Zhou, M. (2020). CodeBERT: A pre-trained model for programming and natural languages. Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., & Zettlemoyer, L. (2020). BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. Proceedings of the 58th Annual Meeting of the Association for

Computational Linguistics, 7871–7880. https://doi.org/10.18653/v1/2020.acl-main.703

Lewis, P., Perez, E., Piktus, A., Karpukhin, V., Goyal, N., Küttler, H., & Riedel, S. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. Advances in Neural Information Processing Systems, 33, 9459–9474. https://arxiv.org/abs/2005.11401

Liu, I., Wang, Y., Zhang, Y., & Neubig, G. (2023). CodeT5+: Open code large language models for code understanding and generation. arXiv preprint arXiv:2305.07922. https://arxiv.org/abs/2305.07922

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., & Stoyanov, V. (2019). RoBERTa: A robustly optimized BERT pretraining approach. arXiv preprint arXiv:1907.11692. https://arxiv.org/abs/1907.11692

Papineni, K., Roukos, S., Ward, T., & Zhu, W. J. (2002). BLEU: A method for automatic evaluation of machine translation. Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, 311–318. https://doi.org/10.3115/1073083.1073135

Ren, S., Guo, D., Lu, S., Zhou, L., Ma, S., Zhou, J., & Li, H. (2020). CodeBLEU: A method for automatic evaluation of code synthesis. arXiv preprint arXiv:2009.10297. https://arxiv.org/abs/2009.10297

Wang, Y., Wang, W., Joty, S., & Hoi, S. C. (2021). CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, 8696–8708. https://doi.org/10.18653/v1/2021.emnlp-main.707