AI-Driven Automated Software Documentation Generation for Enhanced Development Productivity

Conference Paper · July 2024

DOI: 10.1109/ICDSNS62112.2024.10691221

3 authors, including:

Sunil Raj Thota
Amazon
12 PUBLICATIONS **108** CITATIONS
SEE PROFILE

Sandeep Gupta
Samrat Ashok Technological Institute
40 PUBLICATIONS **352** CITATIONS
SEE PROFILE

2024 International Conference on Data Science and Network Security (ICDSNS)

AI-Driven Automated Software Documentation Generation for Enhanced Development Productivity

1st Sunil Raj Thota
Technology, Engineering
Andhra University
Visakhapatnam, India
thotasunilraj@gmail.com

2nd Saransh Arora
Jaypee Institute of information Technology
Noida, India
saransha.1994@gmail.com

3rd Sandeep Gupta
Techieshubhdeep it Solutions Pvt. Ltd,
Gwalior, India
ceo.techies@gmail.com

Abstract—**In the competitive software development industry, effective and superior documentation is a must today. Based on complicated AI models, automated code generation solves this problem and helps produce documentation easily. This work introduces a new approach to automatically generating software documentation, focusing on fine-tuning sophisticated AI models such as GPT-2 and RoBERTa by leveraging a large existing dataset from the GitHub CodeSearchNet challenge. The researchers indicate that RoBERTa outperforms GPT-2 on both accuracy and loss metrics, with an amazing accuracy score of 99.94% vs 74.37% for GPT-2. RoBERTa also demonstrates much lower training and validation losses to highlight its advantages. Another benefit of RoBERTa is its significantly smaller training and validation losses (0.010 and 0.002, respectively) than GPT-2 (1.407 and 1.268). The implication of the above is that quality of documentation and more efficient development are achievable with AI-driven automated documentation production.**

Keywords—Documentation Generation, Software development, Artificial Intelligence (AI), GPT-3, RoBERTa.

I. Introduction

The process of developing software is difficult and takes time. Analysis and coding are its two primary stages [1]. The requirements and software system architecture are established during the analysis phase. Writing and testing source code to satisfy the first phase's requirements takes place during the coding phase. System maintenance is typically incorporated as an extra stage during the cycle of software development, whereas earlier stages can be modified to accommodate evolving system user requirements. A flow diagram for fundamental software development approach is displayed in Fig. 1[1].

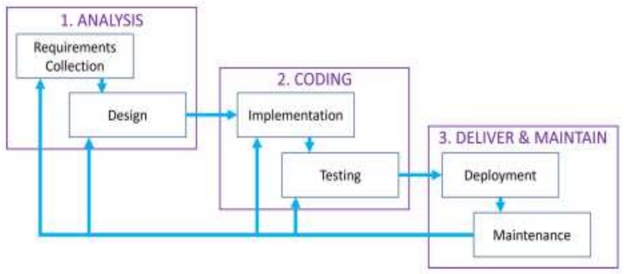

Fig. 1. Fundamental software development approach[1]

Fig. 1. Three phases, each with several steps, make up this example software building model. Compared to the model shown in this flowchart, the number of steps and order of various software development models are different in real-world applications.[1].

Within Software Engineering (SE), developers frequently attempt to ascertain the purpose and usage of a particular code unit (such as a method). It can accomplish this by going through the source code documentation. Effective software development depends on well-written documentation. However, the creation and upkeep of such documentation is expensive and time-consuming. Furthermore, documentation eventually becomes outdated when the system (i.e., code-base) is continuously updated or modified.[2]This study only focuses on automatically creating code-related documentation. Coding documentation needs to be precise, succinct, and unambiguous for the benefit of maintenance process developers. Source code documentation is a crucial procedure for managing and upkeep of software projects. Expert human time and effort are always heavily invested in the documentation process. There should be no ambiguity, and software project documents should be precise and straightforward.[3].

Many strategies have been investigated in Artificial Intelligence (AI) to assist with software development. The emergence of AI-driven code generation has sparked a vibrant debate in the software development community. This technology, which leverages advanced machine learning models like[4][5][6][7]OpenAI's GPT-2, RoBERTa, promises to transform the traditional coding process. Automated software documentation generation powered by AI holds significant promise for revolutionizing development productivity by alleviating the burden of manual documentation tasks. The AI techniques such as machine learning, deep learning as well as the NLP algorithms can go through the codebases effectively and extract the information which is relevant and also, It can produce intelligible documentation automatically. This also reduces the time spent on documenting data automatically and enhances data quality, accuracy, and integrity, significantly minimising errors and inconsistencies. Aside from this, AI-infused documentation creation also contributes to the transfer of knowledge and collaboration among the team members, hence the higher quality of software and development productivity.

It is widely known that implementing the software developers' effective and precise documentation requirements is critical and difficult. The highly recent development of online work due to problems like the COVID-19 epidemic has created an urgent need for technology-based solutions to speed up paperwork processing. The technique aims to meet this demand using Artificial Intelligence (AI) to automate software documentation production. Integrating powerful AI models, like GPT-2 and RoBERTa, revolutionises development efficiency by substantially reducing the time and effort necessary for documentation tasks. Unlike traditional methodologies, AI-driven methodology provides unparalleled accuracy, consistency, and efficiency, opening up opportunities for improved software quality.

A. Paper Contribution

The primary contribution of this study to the field of artificial intelligence automated software documentation production is the application of two prominent AI models, GPT-2 and RoBERTa, which were fined-tuned on a dataset that generated by using Python component from the GitHub CodeSearchNet Challenge. Considering the aim of boosting the project results and enhancing the quality of documents, this study trains and estimates the models using the dataset's richness and flexibility. Automation of the documentation generation process lets developers focus more willingly on coding works as well as gives them perfect and correct documents which leads to less errors and no human efforts. The studies pick up already developed and enhanced models and design a system that measures performance through accuracy and loss indicators. Overall, this study shows great promise for improving software development techniques and increasing efficiency through AI-driven automation of documentation activities. The following points provide the paper contribution of this work:

- The research uses powerful AI models, such as RoBERTa and GPT-2, to automate the process of software documentation development.
- The study assures excellent accuracy and alignment with real-world code and documentation patterns by fine-tuning these models on a large-scale dataset generated from the Python section of the GitHub CodeSearchNet Challenge.
- By minimising manual documentation authoring, developers can increase their efficiency and focus more on coding and creativity.
- To increase the quality and consistency of software documentation by correctly reading code semantics and producing useful documentation, allowing for greater understanding and reliability of codebases.

B. Organization of paper

The following paper arrange as: Section I provide the introduction of the paper with significance, motivation, and research contribution, then Section II provide the related work on this topic, namely automatic software code document generation based on AI models, Next Section III describes the methodology for this document generation and Section IV discuss the results and discussion of the AI models based on accuracy and loss, while last section V provide the conclusion and limitation with future work of this paper.

II. Related Work

This section will examine current research on automatically creating and evaluating software code documents using a variety of tools and methods.

In Khan and Uddin, (2023), using documentation and source code as input, produce code examples using Codex is a model built on the GPT-3 architecture and pre-trained on computer and natural languages. Based on the initial analysis of 40 scikit-learn methods, this methodology produces high-quality code examples: 82.5% of the code samples appropriately addressed the target method and documentation (relevance), while 72.5% of the code examples were performed correctly (passability). Also discovered that passability is further improved from 72.5% to 87.5% by including error logs in the input.[8].

Nassif et al. (2022) present DScribe, a tool-supported method that lets programmers integrate unit test and documentation patterns to produce documentation and tests. DScribe can automatically generate 97% of tests plus documentation for 835 specs, 85% of which were lacking.[9].

Moser and Pichler (2021) describe the creation of six tools that target various issue domains (like engineering, banking, and insurance), programming languages (like COBOL, Java, and C), and SE operations (like maintenance and migration). They conducted an industry case study to assess the platform's efficacy for tool creation. They discussed the findings regarding possibilities for reuse, the adoption of novel languages, and the application of a general-purpose temporary depiction.[10].

Khan and Uddin, (2022), used Codex to generate code documentation automatically. A model built on the GPT-3 architecture, Codex has been trained beforehand in both programming and natural languages. Codex works better than current methods, even in simple configurations like one-shot learning (i.e., training with a single example). With six different programming languages, Codex gets a total BLEU score of 20.6 (an improvement of 11.2% over previous state-of-the-art methods) [2].

Therefore, Xue, (2023), offer methods that will facilitate the development of code generators and increase their reusability. Use information formatted tree-to-tree mappings and apply the "Code Generation by Example" (CGBE) concepts. CGBE application to acquire a UML-to-Although the Java code generator performs well in terms of training dataset size and length[11].

Ren et al., (2023), first carry out an empirical investigation and list the main issues that LLMs face when managing exceptions: try-catch abuse, improper exception handling, and incomplete exception handling. The KPC-based technique has a great deal of promise to improve the code quality produced by LLMs, as evidenced by extensive experimental results. It does this by handling exceptions well, achieving impressive gains of 109.86% and 578.57% with static assessment techniques, and reducing.[12].

In Hashemi, Nayebi and Antoniol, (2020), Examine Stack Overflow Q&As and categorise machine learning documentation Q&As to determine issue kinds, causes, and possible documentation changes. Will utilise findings to improve on state-of-the-art automated documentation generating approaches and expand software functionality adoption, summary, and explanations[13].

In Arthur, (2020), employing the Natural Language Generation methodology, this system is able to properly provide documentation for a C program in addition to user-defined and preset methods. According to comparison results, small and medium-sized software projects can perform better with the suggested system.[3].

Idrisov and Schlippe (2024) evaluate the consistency, maintainability, and correctness of program code produced by AI and humans. While CodeWhisperer was unable to answer any of the 18 challenges, powered by Codex (GPT-3.0), demonstrated best performance, solving 9 of the 18 problems (50.0%). For seven difficulties (38.9%), BingAI Chat (GPT-

4.0) produced right program code; for four problems (22.2%), ChatGPT (GPT-3.5) and Code Llama (Llama 2); and for just one problem (5.6%), StarCoder and InstructCodeT5+. When compared to developing the program code from scratch, there is a time savings of 8.9% to even 71.3% when only minor code changes are required to address the issues raised by 11 AI-generated erroneous codes (8.7%)[14].

In Hu et al., (2022), BLEU, METEOR, ROUGE-L, CIDEr, and SPICE methods employ deep learning to generate code documentation from massive source code corpora. With modest Pearson correlation r approximately 0.7, METEOR correlates well to human assessment measures. However, it is substantially lower than the connection among annotators (with a strong Pearson correlation r around 0.8) and other task correlations described in available research.[15].

In Sajji, Rhazali and Hadi, (2023), automates class diagram generation from source code using Graph Neural Networks (GNNs), an ML technique, in Model Driven Architecture (MDA) and reverse engineering. The suggested method shows how GNNs may automate class diagram creation and improve software development as well as documentation[16].

The research on automatic software code document generation covers methods like NLP-based source code summarization, API documentation systems, tools for reverse engineering, model-driven development methodologies, GPT-3-based code example generation, and tools for API documentation systems. These studies present a variety of strategies meant to raise the correctness and efficiency of code documentation procedures.

III. Methodology

The approach for AI-Driven Automated Software Documentation Generation for Enhanced Development Productivity makes use of a large-scale dataset, notably the Python portion of the CodeSearchNet Challenge dataset on GitHub, which contains 2 million (comment, code) combinations. This dataset was selected because it is widely used in source code and natural language processing studies. Preprocessing included auto-formatting in accordance with the PEP-8 Python style guide, as well as deleting the majority of code comments. Artificial intelligence was used to fine-tune pre-trained models, such as RoBERTa, and GPT-2, on the CodeSearchNet dataset. Tokenization approaches were implemented using Google SentencePiece, which replaced the default Spacy tokenizer. Training made use of the FastAI package, which included techniques such as automated learning rate determination via lr_find and one-cycle policy (fit_one_cycle) to speed up convergence. Model fine-tuning entailed training on top of previously learned models, followed by unfreezing and prolonged training epochs. Additional training information and software specs are available in the related repository and FastAI documentation. To assess the accuracy and train/validation loss of AI models.

A. Data Collection

This study uses the Python part of the 2 million (comment, Python code) pairings in the GitHub CodeSearchNet Challenge dataset.[1]that are drawn from public libraries; this dataset was chosen because of its importance to NLP and source code study. It contains a wide range of Python code samples and comments, giving adequate data for training and assessment.

B. Data Preprocessing

A number of preprocessing procedures were carried out before the data was fed into the models. This involved using the autopep8 tool for automatic formatting in accordance with the PEP-8 Python style guide and deleting code comments. The preprocessing will make it possible to standardize the format of the code snippets and comment out noises, preparing better input for the models.

In addition to auto-formatting and removing code comments rendering to PEP-8 Python style guide, preprocessing steps for AI-driven automated software documentation generation project include tokenisation using Google SentencePiece for subword tokenisation, noise removal to eliminate extraneous characters and symbols, standardisation of code snippet formats, handling special tokens within the code or comments, and data augmentation techniques to enrich the training data. These steps collectively ensure the input data is clean, standardised, and conducive to effective model training for generating accurate and coherent software documentation.

C. Model Selection

For investigation, AI-based models were chosen: mention some wake-sleep models, Transformer architectures—like GPT-2—and the ones under the umbrella of RoBERTa. [17]Transformer-style topologies are used, which offer stable performance in NLP jobs that include generating text and filling in the blanks. To adapt to the specific job of code documentation, the models were further enhanced after pre-training them on huge English text corpora.

1) AI-Based GPT-2 and RoBERTa Models: The following section provide the AI-based GPT-2 and RoBERTa Models for document generation.

a) GPT-2 model[18]: The study "Improving Language Understanding through Generative Pre-Training" by OpenAI introduced the Generative Pre-trained Transformer (GPT) model. According to the authors' publication titled "Language Models are Unsupervised Multitask Learners," OpenAI unveiled the GPT-2 model after attaining this critical juncture. These models have made a major contribution to the field of NLP and have drawn a lot of interest due to their capacity for producing and interpreting language. Though GPT-2 employs a larger dataset for tests, the GPT and GPT-2 have similar structures. There is an abundance of training data for GPT-2. With up to 1.5 billion parameters, OpenAI's GPT-2 was released. Consequently, GPT-2's pre-training architectural options are limited and cannot properly integrate context.[19].

b) RoBERTa model[20]: The RoBERTa model is a substantially enhanced BERT pretraining methodology. While the RoBERTa and BERT models have little differences in structure, the pre-training approaches have been altered. The RoBERTa model surpasses the BERT model regarding training data, batch size, and parameter values. Additionally, BERT's training methodology differs from prior language representation models since it integrates the Masked Language Model (MLM) and Next Sentence

[1] https://github.com/github/CodeSearchNet

Prediction (NSP) techniques. Furthermore, the RoBERTa model employs distinct pre-training techniques. Initially, the NSP duty is removed. It also makes use of dynamic masks. During data preprocessing, a static mask is given to the BERT model. Dynamic masks are used in the RoBERTa model, where several mask modalities are utilized in various data sequences. By using this technique, the RoBERTa model may be trained on a vast quantity of data to learn various masking techniques for various language forms.[19]. In Fig. 2, the RoBERTa model has been organised to transform words into low-dimensional vectors, record text semantic information with the 12-layer transformer, as well as output trained sentence and character vectors via the RoBERTa model. This equation generates text embedding discussed as (1):

$$E_t = E_{token\_emb} + E_{seg\_emb} + E_{pos_{emp}}{}' \dots\dots\dots \quad (1)$$

Et indicates the embedding representation of the t-th character, Etoken_emb represents the token embedding of the character, Eseg_emb provides the segment embedding of the character, as well as Epos_emb defines the position embedding of the character.

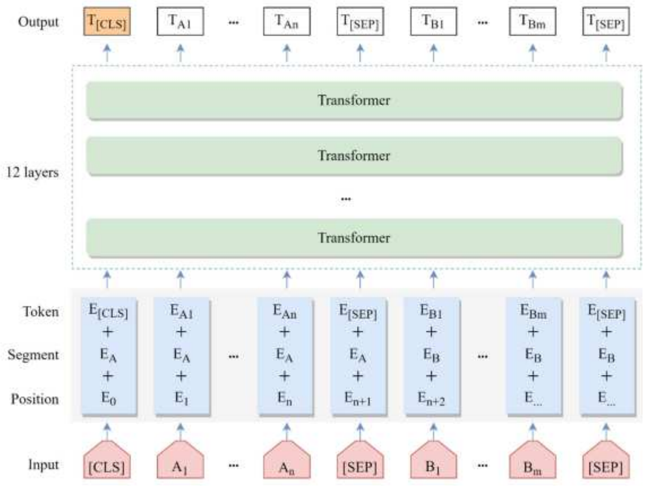

Fig. 2. The model of RoBERTa[21].

Subsequently, the RoBERTa model's middle layer receives the embedding representation and uses an encoder within the 12-layer transformers to extract semantic data. The multi-head attention system, comprised of many self-attention layers, is a crucial component in the transformer. The dot product of attention determines the equivalent output of a single self-attention process; therefore, merging all heads yields multiple self-attention. The computation for multi-head attention can be obtained by using a linear transformation.

Lastly, the following equations may derive self-attention and multi-head attention in (2) and (3):

$$head_i = Attention\left(QW_i^Q, KW_i^K, VW_i^V\right) \dots . \quad (2)$$

$$MultiHead(Q, K, V) = Concat(head_1, head_2, \dots\dots\dots\dots.., head_N)W^O \dots \quad (3)$$

Q, K, and V are the input vectors to the attention mechanism, while WO, WQi, WKi, and WVi are its weight matrices.

Lastly, the RoBERTa model's output layer includes a sentence vector (CLS) that includes global semantic and contextual data. This data may be used to check whether the candidate item's short text and descriptive text have identical semantic contexts.

D. Fine-Tuned of AI models

Model selection for GPT-2 and RoBERTa involves understanding their architectural differences and the fine-tuning process crucial for adapting them to specific tasks like document generation. It is used the AI pre-trained models to refine the models for the chosen tasks with the help of the CodeSearchNet dataset on GitHub. It is used 30 epochs for fine-tuning both models since, as discovered throughout experiments, that's when every model generated its most noticeable improvement. In a similar vein, spent 10 epochs honing the transformers before calling it quits. Fine-tuning typically the involves adjusting hyperparameters and employing specific training strategies to optimize model performance. For GPT-2, hyperparameters like number of epochs 10 to 30 and Average Stochastic Gradient Descent (ASGD) optimizer. At the same time, after 10 epochs of fine-tuning the models, the Transformer networks were trained for one epoch to match the head. It is also possible to use methods like domain-specific data augmentation and transfer learning from models that have been trained Similarly, for RoBERTa, fine-tuning involves hyperparameters. Techniques such as dynamic masking and multi-task learning can also enhance model performance.

E. Model Evaluation

To check how the models handle the performance-in-documentation-generation the models were evaluated in terms of accuracy for produced documentation and the training/validation loss to measure the learning dynamics and the possibility of overfitting.

The research validates itself by filling the gap of efficient and accurate documentary software in the software development sector where remote work is increasingly common. Furthermore, the proposed method employs AI models, such as GPT-2 and RoBERTa, to automatically generate documentation. In this way, the method saves time and effort and guarantees high quality. The stringent experimental procedures and analysis validate the approach and further increase the development process while improving documentation quality.

IV. RESULTS ANALYSIS

The results achieved through machines learning training comprehensively for the chosen AIs such as GPT-2, RoBERTa on CodeSearchNet Challenge dataset. These two AI models show their performance in terms of accuracy, training, and validation loss.

A. Accuracy

The percentage of correctly predicted cases among all the examples in the dataset is known as accuracy, calculated as equation 4. The Accuracy of code documentation is determined by the number of clear documents that explain the purpose or meaning of a given snippet with related code. It offers a broad indicator of how well the model interprets the intended meaning of code.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FN} + \text{FP}} \dots\dots \quad (4)$$

To conduct the measurements, this has employed four variables: "true negative (TN), false negative (FN), true positive (TP), and false positive (FP)". These variables can be measured as follows using target and prediction matrices:

i) TP: forecast positive and it's true.
ii) TN: forecast negative and it's true.
iii) FN: forecast negative and it's false.
iv) FP: forecast positive and it's false.

Training and Validation Loss: Loss functions are used in training and validation to quantify the difference between the values in a dataset that are expected to be there and the actual values. Within the framework of training a model, loss is computed at each training iteration and utilised to adjust the model's parameters in order to reduce errors. The error on training dataset is represented by training loss, and error on a different validation dataset is measured by validation loss.

The following Table I shows the performance of both models, with train and validation losses of 1.40% and 1.26% with the GPT-2 model and RoBERTa training loss of 0.010 and validation loss of 0.002%, respectively. Also, the RoBERTa model achieved 99% accuracy, while GPT-2 achieved only 74% accuracy.

TABLE I. COMPARISON BETWEEN BOTH AI-BASED MODELS FOR DOCUMENT GENERATION

Models	Accuracy	Train Loss	Validation Loss
GPT-2	74.37	1.407	1.268
RoBERTa	99.94	0.010	0.002

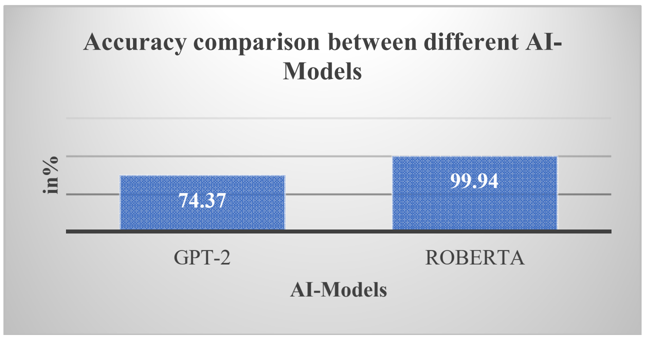

Fig. 3. Bar graph of accuracy performance

Fig. 3 compares the accuracy performance of two artificial intelligence (AI) models for document generation: GPT-2 and RoBERTa. The graph shows that RoBERTa greatly beats GPT-2 in terms of accuracy, with a stated accuracy of 99.94% vs GPT-2's 74.37%. This suggests that RoBERTa is better capable of producing documents with more adherence to intended content or context than GPT-2.

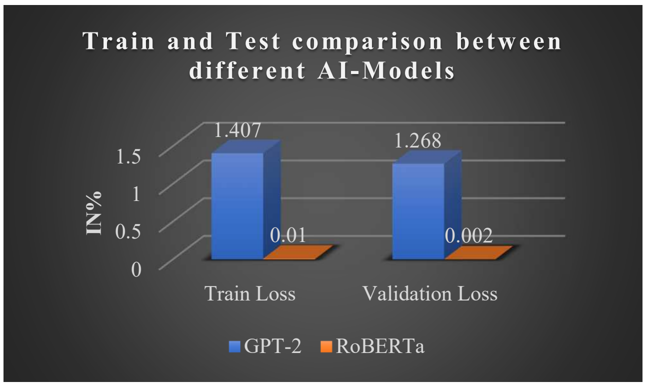

Fig. 4. Bar graph of train and validation loss performance

Fig. 4 represents the combined training success and validation loss performance of both models. The RoBERTa has considerably small value of loss with a training loss of 0.010 and the validation loss of 0.002, which show that the model of training having improved end convergence and better generalization to new data. On the contrary, GPT-2 shows bigger losses with 1.268 and 1.407 the validation and the training, so the performance needs improvement and a higher caution against the overfitting.

On average, these figures explain that RoBERTa beats GPT-2 in terms of accuracy and loss, and hence, RoBERTa would be the perfect AI-driven auto software documentation generator to enhance productivity during production since RoBERTa features higher precision and its capability to generalize much better than GPT-2.

B. *Comparative Analysis*

The following table II shows the accuracy comparison between various AI models (GPT-2[17], CodeNet [22] and RoBERTa) for code generation.

TABLE II. ACUURACY COMPARISON BETWEEN AI-BASED MODELS FOR DOCUMENT GENERATION

Models	Accuracy (%)
GPT-2	74.37
CodeNet	98.5
RoBERTa	99.94

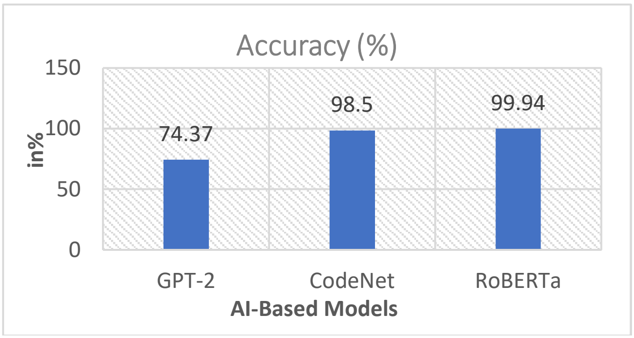

Fig. 5. Bar graph of Accuracy comparison between AI models

There are noticeable differences in the performance of AI-based models used for generating automated software documentation, shows in Fig. 5. GPT-2 achieves a reasonable accuracy of 74.37%, however CodeNet surpasses it greatly with an impressive accuracy of 98.5%. RoBERTa demonstrates excellent accuracy of 99.94%, highlighting its ability to understand complex software environments. RoBERTa's potential to revolutionise automated documentation processes is shown by this comparison. It provides developers with extremely accurate and contextually appropriate documentation, considerably improving development efficiency.

C. Discussion

The research involves applying big data analytics and sophisticated AI models like GPT-2 and RoBERTa to automated software documentation creation. Gaining from the remote work increase, the project proposes enhancing the software documenting with proper data pretreatment and a good model choice. The results provide a great insight into the efficiency differences of the models. The high accuracy of a Roberta 99.94% also outperforms that of a GPT-2, whose score is 74.37%. RoBERTa also minimizes training and validation losses according to which convergence efficiency gets enhanced and generalisation improves. As opposed to BERT, Robert is more precise and adjusts to disparate facts, which is why it will be more suitable for software documentation. Unlike GPT-2 and CodeNet, which have an excellent 98.5% correct predictions, RoBERTa performs better and beats them. It demonstrates that the RoBERTa algorithm can revolutionize documentation automation by giving developers the highly accurate and contextually relevant documentation thus increasing the development productivity. It would be worth mentioning that this study's focus on the Python code and the comments does not present other software development contexts. Although RoBERTa produces good results, more study can be useful to train it better and make it easier to work across different environments. The tests resulted in the fact that RoBERTa can change software documentation creation since it gives developers a very strong tool to apply to process optimization and improve productivity. Developers can augment their software documentation work by employing adequate AI model such as RoBERTa, improving software development procedures.

V. Conclusion And Future Work

This uses GPT-2 and RoBERTa as AI models to generate the software documentation automatically, so trying to fill the vacancy of ensuring efficient and correct documentation in the modern software creation. Rigorous experimentation and analysis were performed on these models initially and the performance of each model was evaluated after being trained on a large-scale dataset from the GitHub CodeSearchNet Challenge. The performance of RoBERTa over GPT-2 is proven: its accuracy (99.94%) very close to the perfect, and the training and validation loss metrics nearly two times lower. This is evidence of the effectiveness of the production practice of RoBERTa by creating documentation aligned with the code patterns existing in the real world to increase productivity and quality of code development. With the research making notable progress in automating documentation generation and confirming the excellent performance of RoBERTa, it is still necessary to look farther to deal with complex code structures and provide the model with the generalist characteristic. In the future, it is imperative to continue improving the accuracy of existing models and assessment techniques so that AI-assisted documentation generation can fulfil its promise of improving software development practices and quality.

References

[1] E. Dehaerne, B. Dey, S. Halder, S. De Gendt, and W. Meert, “Code Generation Using Machine Learning: A Systematic Review,” IEEE Access. 2022. doi: 10.1109/ACCESS.2022.3196347.

[2] J. Y. Khan and G. Uddin, “Automatic Code Documentation Generation Using GPT-3,” in ACM International Conference Proceeding Series, 2022. doi: 10.1145/3551349.3559548.

[3] M. P. Arthur, “Automatic Source Code Documentation using Code Summarization Technique of NLP,” in Procedia Computer Science, 2020. doi: 10.1016/j.procs.2020.04.273.

[4] F. Harrag, M. Dabbah, K. Darwish, and A. Abdelali, “Bert Transformer model for Detecting Arabic GPT2 Auto-Generated Tweets,” ArXiv, vol. abs/2101.0, 2021.

[5] A. S. Hassan Younas, Zohaib Ur Rehman Afridi, Kaleem Ullah, Sahib Gul Afridi, “Development of Photocatalytic Ultrafiltration Membranes Technology for Enhanced Removal of Carbamazepine: Optimization, Mechanistic Insights, and Engineering Applications”.

[6] M. Hämäläinen, K. Alnajjar, and T. Poibeau, “Modern French Poetry Generation with RoBERTa and GPT-2,” Proc. 13th Int. Conf. Comput. Creat. ICCC 2022, no. Veale 2016, pp. 12–16, 2022.

[7] S. G. Kumar et al., “Chronic Reductive Stress Modifies Ribosomal Proteins in Nrf2 Transgenic Mouse Hearts,” Free Radic. Biol. Med., 2022, doi: 10.1016/j.freeradbiomed.2022.10.125.

[8] J. Y. Khan and G. Uddin, “Combining Contexts from Multiple Sources for Documentation-Specific Code Example Generation,” in Proceedings - 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2023, 2023. doi: 10.1109/SANER56733.2023.00071.

[9] M. Nassif, A. Hernandez, A. Sridharan, and M. P. Robillard, “Generating Unit Tests for Documentation,” IEEE Trans. Softw. Eng., 2022, doi: 10.1109/TSE.2021.3087087.

[10] M. Moser and J. Pichler, “Eknows: Platform for Multi-Language Reverse Engineering and Documentation Generation,” in Proceedings - 2021 IEEE International Conference on Software Maintenance and Evolution, ICSME 2021, 2021. doi: 10.1109/ICSME52107.2021.00057.

[11] Q. Xue, “Automating Code Generation for MDE using Machine Learning,” in 2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), 2023, pp. 221–223. doi: 10.1109/ICSE-Companion58688.2023.00060.

[12] X. Ren, X. Ye, D. Zhao, Z. Xing, and X. Yang, “From Misuse to Mastery: Enhancing Code Generation with Knowledge-Driven AI Chaining,” in 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2023, pp. 976–987. doi: 10.1109/ASE56229.2023.00143.

[13] Y. Hashemi, M. Nayebi, and G. Antoniol, “Documentation of Machine Learning Software,” in SANER 2020 - Proceedings of the 2020 IEEE 27th International Conference on Software Analysis, Evolution, and Reengineering, 2020. doi: 10.1109/SANER48275.2020.9054844.

[14] B. Idrisov and T. Schlippe, “Program Code Generation with Generative AIs,” Algorithms, vol. 17, no. 2, p. 62, 2024, doi: 10.3390/a17020062.

[15] X. Hu, Q. Chen, H. Wang, X. Xia, D. Lo, and T. Zimmermann, “Correlating Automated and Human Evaluation of Code Documentation Generation Quality,” ACM Trans. Softw. Eng. Methodol., 2022, doi: 10.1145/3502853.

[16] A. Sajji, Y. Rhazali, and Y. Hadi, “A methodology of automatic class diagrams generation from source codeusing Model-Driven Architecture and Machine Learning to achieve Energy Efficiency,” in E3S Web of Conferences, 2023. doi: 10.1051/e3sconf/202341201002.

[17] J. Cruz-Benito, S. Vishwakarma, F. Martin-Fernandez, and I. Faro, “Automated Source Code Generation and Auto-Completion Using Deep Learning: Comparing and Discussing Current Language Model-Related Approaches,” AI, 2021, doi: 10.3390/ai2010001.

[18] Radford Alec, Wu Jeffrey, Child Rewon, Luan David, Amodei Dario, and Sutskever Ilya, “Language Models are Unsupervised Multitask Learners | Enhanced Reader,” OpenAI Blog, 2019.

[19] H. Zhang and M. O. Shafiq, “Survey of transformers and towards ensemble learning using transformers for natural language processing,” J. Big Data, vol. 11, no. 1, p. 25, 2024, doi: 10.1186/s40537-023-00842-0.

[20] S. V. Liu Y, Ott M, Goyal N, Du J, Joshi M, Chen D, Levy O, Lewis M, Zettlemoyer L, “RoBERTa: a robustly optimized BERT pretraining approach,” CoRR, 2019.

[21] L. Gao, L. Zhang, L. Zhang, and J. Huang, “RSVN: A RoBERTa Sentence Vector Normalization Scheme for Short Texts to Extract Semantic Information,” Appl. Sci., 2022, doi: 10.3390/app122111278.

[22] R. Puri et al., “CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks,” no. NeurIPS, pp. 1–13, 2021.