Journal of Information Systems Engineering and Management
2025, 10(23s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

Analyzing ChatGPT's Problem-solving Capabilities in Arabic-Based Software tasks

Hani Al-Bloush, Laith Al Shehab, Ashraf A.Odeh

Faculty of Information Technology, Middle East University, Amman, 11831, JORDAN
halbloush@meu.edu.jo

, *Faculty of Information Technology, Middle East University, Amman, 11831, JORDAN*

Lshehab@meu.edu.jo

, *Faculty of Information Technology, Middle East University, Amman, 11831, JORDAN*
Aodeh@meu.edu.jo

Corresponding author: halbloush@meu.edu.jo

ARTICLE INFO

Received: 16 Dec 2024

Revised: 02 Feb 2025

Accepted: 20 Feb 2025

ABSTRACT

This study investigates the problem of ChatGPT's performance in generating and optimizing Python code in both English and Arabic, addressing the challenges of low-resource languages. The objective is to compare how ChatGPT versions 3.5 and 4.0 handle standard algorithms-Linear Search, Binary Search, and Quick Sort in terms of code generation, optimization, and readability. Using a comparative experimental design, key performance metrics such as time complexity, execution speed, and error rates were analyzed. The results reveal substantial disparities between the two languages: English exhibited efficient code generation, minimal errors, and improved optimization, while Arabic encountered higher error rates, slower execution, and limited performance gains despite optimization. These findings highlight the limitations of AI models in low-resource linguistic environments, underscoring the need for fine-tuning to enhance global applicability. This study contributes to advancing the understanding of AI coding tools and their ability to support diverse linguistic contexts, particularly in underrepresented languages like Arabic.

Keywords: ChatGPT, Software Problem-solving, Arabic text, Natural Language Processing.

1.INTRODUCTION:

AI-based code generation tools, such as ChatGPT, have gained significant attention for streamlining coding tasks. The evolution of AI in coding can be traced back to early attempts at automating code generation in the 1990s, with basic rule-based systems that provided limited assistance to developers [1]. With advancements in machine learning and the availability of vast datasets, AI has evolved to support more complex tasks, leading to the development of largescale language models [2]. The introduction of models like OpenAI's GPT series marked a significant leap in AI's ability to understand and generate natural language text, including code. GPT-3, in particular, showcased the potential of using AI to assist in various coding-related activities [3]. GPT-3's success paved the way for more advanced iterations like GPT-3.5 and GPT-4, which exhibit even greater proficiency in understanding user prompts and generating accurate code snippets [4, 5].

These developments have positioned tools like ChatGPT at the forefront of AI-assisted coding solutions. These tools use largescale language models trained on vast amounts of text data to assist developers in generating code snippets, debugging, and providing documentation [6, 7]. Tools like Code-Compose have demonstrated the practical impact of LLMs on an industrial scale. However,

challenges persist, particularly around code security, as AI-generated code may introduce bugs or vulnerabilities requiring human validation. Additionally, the performance of models like ChatGPT and Bing AI varies depending on specific use cases, underscoring the need for contextual evaluation [8].

The advancements in artificial intelligence (AI) and natural language processing (NLP) hold immense potential to address global disparities in access to technology and education, aligning with the Sustainable Development Goals (SDGs)[9]. Specifically, SDG 4 (Quality Education) emphasizes the need for inclusive and equitable education, which can be achieved by democratizing access to programming literacy through AI tools like ChatGPT. By improving ChatGPT's capabilities in low-resource languages such as Arabic, this research facilitates the creation of localized educational resources and tools that empower underrepresented communities to participate in the digital economy[10]. Furthermore, SDG 9 (Industry, Innovation, and Infrastructure) underscores the importance of fostering innovation and building resilient technological infrastructures [11]. This study fills a critical gap in the existing literature by addressing the underperformance of AI tools like ChatGPT in low-resource languages, particularly Arabic, a challenge that has been underexplored in prior studies. By systematically evaluating and comparing GPT-3.5 and GPT-4, this study not only provides a nuanced understanding of their capabilities in multilingual contexts but also highlights the specific challenges posed by low-resource languages in code generation, optimization, and readability. This study addresses the technological gaps in low-resource linguistic environments by enhancing AI's ability to generate, optimize, and debug code in Arabic. Such advancements support localized innovation and contribute to a more inclusive and equitable global technology landscape.

Current research heavily emphasizes the capabilities of these models in high-resource languages like English, which benefit from extensive programming literature and datasets [12]. While GPT-3, GPT-3.5, and GPT-4 have made significant advancements in high-resource languages, adapting these models for low-resource languages such as Arabic has proven challenging. Zhang et al. (2023) [13] demonstrated that these models effectively support developers in high-resource environments by generating accurate code snippets and facilitating API exploration. However, their application in low-resource languages reveals significant performance gaps. Lin, Murakami, and Ishida (2020) [14] emphasized that the lack of high-quality training data and coding repositories in languages like Arabic significantly hampers developer productivity. Research has explored strategies like transfer learning and zero-shot modeling to adapt models to new tasks without explicit training, but these approaches, primarily focused on natural language generation (NLG), have not fully addressed the complexities in code generation for low-resource languages [15]. Sontakke et al. (2023) found that transferring coding models from high-resource to low-resource languages yields only limited improvements, suggesting that the inherent complexity of Arabic and its lack of comprehensive datasets remain significant obstacles[16].

This focus on English overlooks challenges faced in coding tasks conducted in lower-source languages like Arabic, where limited data availability affects AI models' performance [17]. The effectiveness of ChatGPT in code optimization and readability presents additional challenges in low-resource languages. Liu et al. (2023) noted that in high-resource languages, careful prompt crafting enhances ChatGPT's ability to generate optimized code snippets [18]. However, these techniques alone do not fully address the unique challenges in Arabic coding contexts. further studies observed that ChatGPT's performance in generating simple and readable code in Arabic impacts long-term project sustainability. Additionally, debugging and defect detection are particularly problematic. Yan et al. (2023) and Liu et al. (2023) found that when tackling complex coding problems in Arabic, ChatGPT often produces incorrect outputs that require extensive human intervention [18, 19]. Peng et al. (2023) highlighted that ChatGPT's translation capabilities in non-English languages are hindered by insufficient training data, creating challenges in debugging and refining code in Arabic[20]. Robinson et al. (2023) underscored these issues, emphasizing ChatGPT's struggles with code optimization and debugging in low-resource languages[21].

Early versions of ChatGPT exhibited varying levels of success depending on the specificity and clarity of user prompts, which spurred research into how prompt design could be optimized for better outcomes [22]. Yet, there is limited research exploring ChatGPT's effectiveness in lower-resource languages like Arabic, despite its growing importance in the global software development community.

The recurring underperformance of ChatGPT in Arabic underscores the need for focused research to improve its capabilities in low-resource languages. Al-Thubaity et al. (2023) [23] demonstrated that ChatGPT has limited capabilities in generating high-quality dialectal Arabic text, while smaller models fine-tuned for Arabic often outperform ChatGPT in producing readable and maintainable code. These findings highlight the limitations of ChatGPT in handling various aspects of Arabic coding and underline the need for optimization in these environments. Although these studies outline general challenges, no comprehensive research has directly compared GPT-3.5 and GPT-4 in Arabic-specific coding tasks, a gap this study aims to address.

Although GPT-3.5 and GPT-4 have been extensively compared in high-resource languages, limited research exists on their performance in Arabic coding contexts. Alyafeai et al. (2023) [24] demonstrated that GPT-4 outperforms GPT-3.5 in several Arabic natural language processing tasks, laying the groundwork for further exploration in software-related applications. This study aims to directly compare GPT-3.5 and GPT-4 in generating Python code for standard algorithms in Arabic, utilizing new performance metrics specifically tailored to low-resource language contexts. These metrics will provide a nuanced evaluation of how effectively these models handle the unique challenges posed by Arabic's linguistic features, addressing broader issues in low-resource language coding.

Early evidence suggests that ChatGPT underperforms in Arabic natural language processing tasks compared to smaller models tailored for Arabic. This raises critical questions about ChatGPT's ability to handle code generation, repair, and summarization in Arabic, a language that presents unique challenges due to its complex structure and limited coding literature. Current evaluations of AI models often prioritize functional code generation while neglecting aspects like code optimization and simplicity. Although ChatGPT shows potential in creating modular and efficient code, challenges remain in handling complex coding scenarios, indicating a need to explore how ChatGPT balances tradeoffs between code simplicity, readability, and performance. Moreover, the emergence of newer ChatGPT versions, such as GPT-3.5 and GPT-4, has introduced enhanced capabilities and expanded use cases for AI-based coding tools [25]. While there are indications of improvements in newer versions, comprehensive comparisons between GPT-3.5 and GPT-4 in coding contexts, particularly in Arabic, are sparse. Understanding how these versions perform in generating, optimizing, and rewriting code in Arabic remains unclear and is a key focus of this study [26, 27].

Additionally, the literature points to maintainability issues in ChatGPT-generated code, emphasizing the importance of readability for long-term code management [28]. Yet, the extent to which ChatGPT can streamline code while preserving performance remains unclear, necessitating further exploration. This study aims to fill this gap by evaluating how ChatGPT performs in coding tasks within a low-resource linguistic environment, with a focus on Arabic. Addressing this gap is crucial for improving AI coding tools' accessibility and utility worldwide, particularly for developers working in diverse linguistic contexts. The findings from this study could serve as a foundation for developing more effective multilingual AI tools by identifying actionable strategies to improve their performance in underrepresented languages. By addressing both technical and linguistic challenges, this research can inform the creation of more inclusive, scalable AI coding solutions tailored to diverse linguistic needs.

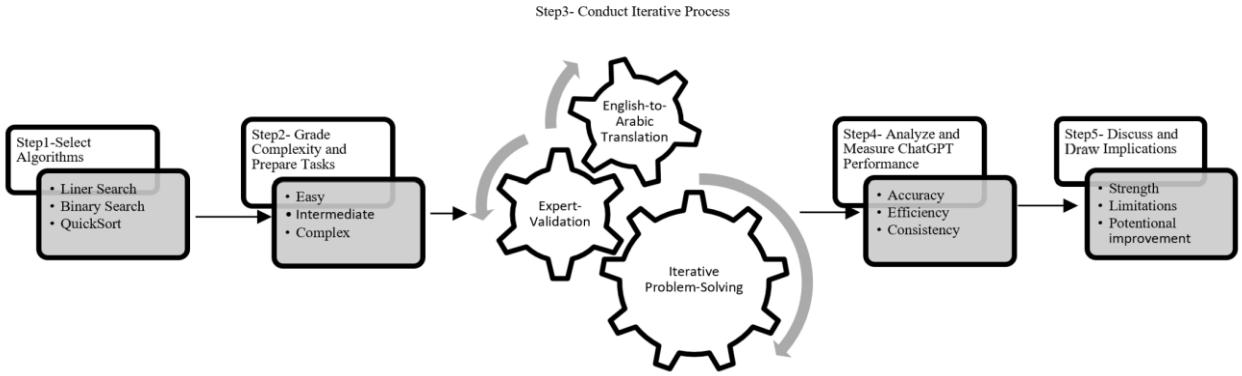
These gaps lead to the following key research questions:

- R1: How effective is ChatGPT (versions 3.5 and 4.0) in generating Python code for standard algorithms, and how do these versions compare in terms of time complexity and execution speed?
- R2: What are the differences in ChatGPT's performance when generating and optimizing code in English versus Arabic, and what impact do these differences have on coding efficiency?
- R3: How does ChatGPT balance code simplicity, readability, and performance when prompted to optimize or rewrite code more concisely?

2. MATERIAL AND METHODS:

This study evaluates ChatGPT's ability to solve software problems with a focus on Arabic language processing. Using an iterative approach [29, 30], the study assesses ChatGPT's performance in code

generation, repair, and summarization. The process is outlined in five key steps, as depicted in Figure 1.



Three Python Algorithms-Linear Search, Binary Search, and Quick-Sort-were chosen to represent varying levels of complexity, from easy to complex. The tasks were translated into Arabic and validated through expert feedback. Errors were identified, logged, and corrected through multiple iterations. Performance was measured using key metrics such as accuracy, precision, recall, and code readability to evaluate ChatGPT's effectiveness in these tasks. The study concludes by discussing ChatGPT's strengths and limitations in handling Arabic-based software problems, offering insights into potential improvements for its application in low-resource languages. The iterative process, supported by expert feedback, ensures a comprehensive evaluation of ChatGPT's capabilities.

2.1. Task Complexity Grading:

The selected algorithms are classified into three complexity levels-easy, intermediate, and complex to evaluate ChatGPT's adaptability across varying tasks. The selection of these algorithms is specifically tailored to assess ChatGPT's performance in low-resource coding contexts, such as Arabic, by examining how well the model handles diverse linguistic and computational challenges.

- Linear Search: Classified as easy, this algorithm iterates through a list to find a target value. With a linear time, complexity of $0(n)$, it is simple and suitable for beginners [31]. In the context of Arabic, Linear Search serves as a baseline for testing ChatGPT's ability to handle straightforward algorithms while accounting for Arabic-specific challenges, such as accurate syntax generation and semantic clarity in translated prompts.
- Binary Search: An intermediate-level algorithm, binary search requires an ordered list and uses a divide-and-conquer approach. It offers a time complexity of O (log n), making it more efficient than linear search for large datasets [31]. This algorithm is particularly relevant to Arabic because it tests ChatGPT's ability to manage hierarchical logic and ordered data structures in a linguistically complex setting, where accurate interpretation of sorted data and conditional logic may be influenced by language-specific nuances.
- Quick-Sort: This complex algorithm employs a divide-and-conquer strategy to sort lists. It has an average time complexity of O (n log n), though it can degrade to O(n^2) in the worst case due to its recursive partitioning technique [31]. Quick-Sort evaluates ChatGPT's proficiency in handling advanced computational tasks involving recursion and modularization, both of which are challenging in low-resource languages like Arabic due to their linguistic complexity and limited training data. Recursive algorithms, in particular, provide insight into the model's ability to generate efficient and readable code in such contexts.

By including these three algorithms, the study captures a comprehensive range of computational tasks to systematically explore ChatGPT's performance in addressing the unique linguistic and computational challenges of low-resource languages, such as Arabic. This selection ensures that the findings are representative of real-world coding scenarios encountered in underrepresented linguistic environments. These algorithms form the basis for systematically exploring ChatGPT's performance in software problem-solving.

2.2. Iterative Interaction with ChatGPT (Revised with Additions for Feedback Cycle)

This step involves an iterative process of translation, expert validation, task processing, and error analysis. The experimental design is uniquely original, developed to address the linguistic and computational challenges posed by low-resource languages like Arabic. Unlike conventional frameworks used for high-resource languages, this design incorporates innovative adjustments that account for Arabic's structural complexity and linguistic nuances.

Arabic instructions are designed with domain-specific terminology to ensure clarity and avoid ambiguities. The process combines translation refinement, linguistic validation, and iterative prompt adjustments, reflecting a pioneering approach to evaluating AI performance in underrepresented languages. To tackle issues such as structural complexity, lack of direct equivalences for technical terms, and dialectal variations, domain-specific terminology was iteratively refined and validated by experts. This ensured that prompts were both linguistically accurate and computationally viable.

The translation phase, facilitated by ChatGPT, was followed by expert review to ensure fluency, coherence, and technical correctness. Translation accuracy was improved through iterative modifications until the desired precision was achieved. Additionally, error analysis accounted for linguistic nuances affecting Arabic programming syntax, including adjustments for deviations from standard programming conventions caused by literal translations. This adaptation helped identify and address model limitations in Arabic-specific contexts, improving its coding performance.

Systematic error analysis tracked various error types (syntax, logic, runtime), logging details such as the prompt, error encountered, and iterations needed for correction. The framework specifically focused on errors unique to low-resource languages, such as ambiguities in linguistic constructs and difficulties in interpreting Arabic-specific task descriptions. By identifying these errors, the framework provided actionable insights into enhancing AI models for underrepresented linguistic environments.

A scoring system evaluated the severity of errors, and expert feedback guided adjustments to reduce mistakes. The iterative process continued for a maximum of five cycles, with performance metrics (accuracy, readability) monitored until a predefined threshold (90% correctness) was reached or improvements plateaued. This threshold accounted for the additional complexity of achieving high accuracy in low-resource languages, where sparse datasets and linguistic diversity required more iterations than typically needed for high-resource languages. Performance tracking was logged using version control systems to document code evolution and measure ChatGPT’s progress over time**.**

2.3. Analysis and Performance Metrics

The outputs ChatGPT generates in response to Arabic instructions in Step 3 are analyzed to assess the accuracy and reliability of these responses within the original software-related tasks. To accomplish this, a set of specific performance metrics is employed, providing a robust framework to measure ChatGPT's aptitude in solving Arabic-based software problems, including code generation, repair, and summarization. The metrics include accuracy, precision, recall, and a detailed grading rubric, which allow a comprehensive assessment of ChatGPT's performance across different tasks.

Code Generation Accuracy: The accuracy of the code generated by ChatGPT will be benchmarked using automated code testing frameworks, such as unit tests, to validate the correctness of the generated code against expected outcomes. The tests will cover edge cases, typical use cases, and error handling to ensure comprehensive evaluation. Additionally, accuracy will be assessed based on the logical structure of the generated code in alignment with the algorithm's intended functionality.

2.4. Code Repair Evaluation (Precision and Recall):

Clarifying Use of Precision and Recall: While precision and recall are typically used in classification tasks, they can be adapted to code repair by measuring the correctness of ChatGPT's responses. In this context:

Precision will be calculated as the ratio of correct code fixes suggested by ChatGPT to the total number of suggestions provided. It measures how many of the suggested repairs are actually correct. Recall will be used to measure the proportion of errors in the original code that ChatGPT successfully identifies and rectifies. It reflects how many of the total existing errors were correctly addressed by ChatGPT. These metrics will be benchmarked against a set of predefined correct solutions vetted by experts to ensure consistency and relevance in the evaluation.

2.5. Code Readability and Complexity (Grading Rubric):

A detailed grading rubric will assess the readability of ChatGPT-generated code, focusing on criteria such as adherence to coding standards (consistent naming, variable use, PEP 8 compliance), effective commenting, logical code structure, and modularization for maintainability. Code complexity will be measured using cyclomatic complexity to assess independent paths through the code. The goal is to ensure the code is both simple and maintainable. Summarization quality will be evaluated based on precision and recall, ensuring key elements are captured concisely. Experts will score the code and summaries on clarity, conciseness, and relevance, providing a quantitative assessment.

3. EXPERIMENTS

3.1. Experiment 1: English Coding Performance on ChatGPT 3.5

We prompted ChatGPT to generate Python code for three algorithms-Linear Search, Binary Search, and Quick Sort-using the standardized phrase: "Write a simple Python code for [Algorithm Name]." The experiment was conducted with Python 3.9 using default settings to replicate general-use conditions. The generated code was evaluated using a grading rubric based on readability, including criteria such as commenting, code structure, adherence to Python's PEP 8 standards, and modularization. The time complexity of each algorithm was then analyzed based on the generated code.

Table 1: Result for Query Number One

Algorithm	Time Complexity
Linear Search	O(n)
Binary Search	O (log n)
Quick Sort	O (n log n) on average, $O(n^2)$ in the worst case

Experiment Evaluation: Binary Search was the most efficient algorithm due to its logarithmic complexity. The generated code adhered to proper commenting and Python standards, showing good readability and modularization. The time complexity of each algorithm was then analyzed based on the generated code, as shown in Table 1.

3.2. Experiment 1: Performance of ChatGPT 4.0 in english Coding Tasks

The experiment on ChatGPT 4.0 used the prompt: "Provide a simple Python code for [Algorithm Name]," ensuring consistency with previous trials. Python 3.9 was used for evaluation, and the generated code was reviewed using a grading rubric assessing readability, including criteria such as commenting, structure, adherence to coding standards, and proper modularization. For Linear Search, the time complexity was O(n), as the algorithm iterates through the list until the target element is found.

Binary Search resulted in a more complex time evaluation with a combined complexity of O(n log n + m log m + n log m), influenced by sorting and binary search operations. Quick Sort had a time complexity of O(n log n), attributed to its recursive, divide-and-conquer strategy. Table 2 provides a summary of the time complexities for these algorithms. The generated code demonstrated effective readability, with proper use of comments, adherence to Python's best practices, and a well-structured, modularized approach.

Table 2: Result for Query Number One

Algorithm	Time Complexity
Linear Search	O(n)
Binary Search	O (n log n + m log m + n log m)
Quick Sort	O (n log n) on average, $O(n^2)$ in the worst case

3.3. Experiment 1: Performance of ChatGPT 3.5 in Arabic Coding Tasks

We prompted ChatGPT in Arabic to generate Python code for three algorithms: Linear Search, Binary Search, and Quick Sort, to solve a document similarity problem. For Linear Search, the prompt used was: "برنامج بسيط باستخدام لغة بايثون لخوارزمية البحث الخطي لحل مشكلة فحص تشابه المستندات," resulting in a time complexity of O(m + n), where m and n represent the lengths of the documents. This algorithm utilized set operations to calculate document similarity, producing code that adhered to Python standards with proper comments and modular functions. Binary Search was prompted with: " برنامج بسيط باستخدام لغة بايثون لخوارزمية البحث الثنائي لحل مشكلة فحص تشابه المستندات," yielding a time complexity of O(log m), where m denotes the number of words in the document. The algorithm operated on a sorted list of words, and the generated code demonstrated clear variable usage, proper structuring, and comprehensive comments.

For Quick Sort, the prompt: " برنامج بسيط باستخدام لغة بايثون لخوارزمية الفرز السريع لحل مشكلة فحص تشابه المستندات" resulted in a time complexity of O(n log n) on average, with a worst-case complexity of $O(n^2)$. The implementation followed the standard Quick-Sort approach with modularization and detailed commenting to enhance readability. Table 3 summarizes the time complexities for the three algorithms. Binary Search was evaluated as the most efficient algorithm due to its O(log m) complexity. Overall, the generated code met readability standards with clear comments, good structure, and proper modularization.

Table 3: Arabic Result for Query Number One

Algorithm	Time Complexity
Linear Search	O (m + n), where m and n are the lengths of the input documents
Binary Search	O (log m), where m is the number of words in the document
Quick Sort	O (n log n) on average, $O(n^2)$ in the worst case

3.4. Experiment 1: Performance of ChatGPT 4.0 in Arabic Coding Tasks

We prompted ChatGPT in Arabic to generate Python code for Linear Search, Binary Search, and Quick Sort to solve document similarity problems. For Linear Search, the prompt was: " برنامج بسيط باستخدام لغة بايثون لخوارزمية البحث الخطي لحل مشكلة فحص تشابه المستندات," and it resulted in a time complexity of O(n). The algorithm calculates cosine similarity between the target and each document, iterating through the collection linearly. For Binary Search, the prompt: " برنامج بسيط باستخدام لغة بايثون لخوارزمية البحث الثنائي لحل مشكلة فحص تشابه المستندات" yielded a time complexity of O(n log n), as the algorithm sorts similarities in descending order and uses binary search to efficiently find matches above the threshold.

Quick Sort, prompted with: " برنامج بسيط باستخدام لغة بايثون لخوارزمية الفرز السريع لحل مشكلة فحص تشابه المستندات," also had a time complexity of O(n log n), where the algorithm applied quick sort to organize similarities, followed by binary search to find relevant matches. Table 4 summarizes the time complexities of the algorithms generated by ChatGPT 4.0 in response to Arabic prompts. Linear Search operates with a linear time complexity, while Binary Search and Quick Sort exhibit logarithmic complexities, making them more efficient for larger datasets.

Table 4: Arabic Result for Query Number One (ChatGPT 4.0)

Algorithm	Time Complexity
Linear Search	O(n)
Binary Search	O (log n)
Quick Sort	O (n log n) on average

4. RESULTS

This section presents the statistical outcomes of experiments conducted using ChatGPT 3.5 and 4.0 for generating, optimizing, and minimizing Python code for three algorithms: Linear Search, Binary Search, and Quick Sort. The results are summarized in tables showing lines of code and execution time for each query type (initial, optimized, and concise) in both English and Arabic.

4.1 ChatGPT 3.5 Experiments

The experiments on ChatGPT 3.5 evaluated its performance in generating and optimizing Python code for three algorithms: Linear Search, Binary Search, and Quick Sort. The results were analyzed based on query types, lines of code, and execution time to assess the efficiency and impact of optimization. When tasked with generating Python code in Arabic, ChatGPT 3.5 faced distinct challenges arising from the structural and linguistic complexities of the language. For example, the accuracy of translations was often impacted by literal interpretations of terms, leading to outputs that occasionally required human intervention for correction. Arabic's unique syntactic structure and lack of direct equivalence for certain technical terms posed additional difficulties, resulting in patterns of errors such as mismatched variables or unclear loop constructions. These challenges were most pronounced in tasks involving algorithms like Quick Sort, where recursion and partitioning logic required precise terminology.

Table 5: Results of ChatGPT 3.5 for English Linear Search, Binary Search, and Quick Sort.

Algorithm	Query Type	Lines of Code	Execution Time (seconds)
Linear Search	Initial Query	9	0.002
	Optimized Query	9	0.003
	Concise Query	2	0
Binary Search	Initial Query	18	0.367
	Optimized Query	18	0.412
	Concise Query	18	0.005
Quick Sort	Initial Query	8	0.002
	Optimized Query	14	0.008
	Concise Query	8	0.003

Table 5 summarizes ChatGPT 3.5's results for generating and optimizing Python code in English. Execution times vary slightly, especially for Binary Search, while concise versions reduce code length and improve execution time for Linear Search, highlighting optimization.

Table 6: Results of ChatGPT 3.5 for Arabic Linear Search, Binary Search, and Quick Sort.

Algorithm	Query Type	Lines of Code	Execution Time (seconds)
Linear Search	Initial Query	12	0.002
	Optimized Query	12	0.003
	Concise Query	7	0
Binary Search	Initial Query	18	0.367
	Optimized Query	19	0.005
	Concise Query	18	0.005
Quick Sort	Initial Query	8	0.002
	Optimized Query	12	0.008
	Concise Query	8	0.003

Table 6 highlights the results of Arabic code generation tasks, where issues such as inconsistent handling of Arabic-specific input and unclear error messages emerged during the testing process. For example, in the case of Binary Search, the model occasionally misinterpreted sorted Arabic inputs, affecting its ability to correctly execute comparisons. This highlights the need for iterative prompt adjustments and enhanced Arabic-specific datasets to mitigate these errors. Additionally, Linear Search tasks saw increased lines of code due to ChatGPT 3.5's inability to condense repetitive logic effectively in Arabic compared to its English counterpart.

4.2 ChatGPT 4.0 Experiments

The experiments on ChatGPT 4.0 focused on its ability to generate and optimize Python code for Linear Search, Binary Search, and Quick Sort in both English and Arabic. The results were analyzed for query types, code length, and execution time, highlighting ChatGPT 4.0's advancements in optimization and efficiency across languages. A notable observation is the stark differences in execution speeds between English and Arabic outputs. For example, while Linear Search in English exhibited an execution time of 0.003 seconds in its initial query, the same task in Arabic required 2.335 seconds, as shown in Table 8. These disparities are largely attributable to the inherent complexities of processing Arabic text, such as its right-to-left script and the additional computational overhead needed to handle structural and syntactic nuances. Similarly, Binary Search and Quick Sort in Arabic also demonstrated longer execution times compared to their English counterparts, despite comparable lines of code in some cases.

Table 7: Results of ChatGPT 4.0 for English Linear Search, Binary Search, and Quick Sort.

Algorithm	Query Type	Lines of Code	Execution Time (seconds)
Linear Search	Initial Query	17	0.003
	Optimized Query	16	0.004
	Concise Query	13	0.009
Binary Search	Initial Query	26	0.007
	Optimized Query	21	0.005
	Concise Query	13	0.008
Quick Sort	Initial Query	21	0.008
	Optimized Query	16	0.004
	Concise Query	7	0.003

Table 7 shows ChatGPT 4.0's results for generating and optimizing Python code in English for Linear Search, Binary Search, and Quick Sort. Linear Search shows reduced lines of code but a slight increase in execution time. Binary Search demonstrates a significant reduction in code length with minor improvements in execution time. Quick Sort optimizes both code length and execution time, reflecting ChatGPT 4.0's ability to streamline code across algorithms. When addressing linguistic complexities, ChatGPT 4.0 exhibited notable improvements over 3.5 but still encountered similar challenges. For instance, while Arabic outputs for Quick Sort showed better modularization and fewer syntax errors than 3.5, structural mismatches and overly literal translations persisted in tasks like Binary Search. These issues often stemmed from the model's inability to adapt pre-trained coding conventions for high-resource languages to Arabic's unique linguistic features. Despite these challenges, ChatGPT 4.0 demonstrated superior adaptability in Arabic-specific contexts, particularly in reducing redundant logic and improving readability across optimized queries.

Table 8: Results of ChatGPT 4.0 for Arabic Linear Search, Binary Search, and Quick Sort.

Algorithm	Query Type	Lines of Code	Execution Time (seconds)
Linear Search	Initial Query	27	2.335
	Optimized Query	23	1.892
	Concise Query	11	0.005
Binary Search	Initial Query	35	1.078
	Optimized Query	36	0.002
	Concise Query	28	0.004
Quick Sort	Initial Query	41	0.001
	Optimized Query	36	0.003
	Concise Query	25	0.004

Table 8 summarizes ChatGPT 4.0's performance in Arabic code generation for Linear Search, Binary Search, and Quick Sort. Linear Search shows a significant reduction in execution time and lines of code in the concise query. Binary Search improves execution time (0.002 seconds) despite a slight increase in code length. Quick Sort consistently reduces both lines of code and execution time, highlighting ChatGPT 4.0's ability to optimize performance across tasks. These findings suggest that ChatGPT 4.0's ability to handle Arabic linguistic complexities was enhanced compared to 3.5, but persistent issues with translation accuracy and computational overhead underscore the need for further refinement in Arabic-specific datasets and model tuning.

4.3 Comparative Summary

The comparative analysis highlights the performance of ChatGPT 3.5 and 4.0 in generating concise Python code for Linear Search, Binary Search, and Quick Sort across English and Arabic. The results examine lines of code and execution time to evaluate trade-offs between conciseness and performance across the two versions. In particular, Arabic coding tasks introduced unique challenges not present in English tasks, such as handling linguistic nuances, structural mismatches, and computational overhead due to the complexity of Arabic syntax. These challenges often resulted in longer execution times and increased lines of code for Arabic tasks compared to their English counterparts.

Table 9: Comparative summary of concise versions for ChatGPT 3.5 and ChatGPT 4.0 (in English and Arabic).

Algorithm	Version	Language	Lines of Code	Execution Time (seconds)
Linear Search	3.5	English	2	0
	4	English	13	0.009
	3.5	Arabic	7	0
	4	Arabic	11	0.005
Binary Search	3.5	English	18	0.005
	4	English	13	0.008
	3.5	Arabic	18	0.005
	4	Arabic	28	0.004
Quick Sort	3.5	English	8	0.003
	4	English	7	0.003
	3.5	Arabic	8	0.003
	4	Arabic	25	0.004

The trends observed indicate that ChatGPT 4.0 generally produced more concise code, particularly for Quick Sort, but at times this conciseness came at the expense of execution speed, especially for Arabic tasks. For example, while Binary Search in Arabic demonstrated improved modularization and readability under ChatGPT 4.0, it also saw a significant increase in lines of code compared to 3.5. This suggests that the improvements in code quality and clarity made by 4.0 required additional computational resources in certain scenarios.

Another notable trend is the improvement in ChatGPT 4.0's ability to handle linguistic complexities, which allowed it to generate more syntactically accurate Arabic code compared to 3.5. However, these improvements were not uniform across all tasks. For Linear Search, while 4.0 produced more structured code in Arabic, it required slightly more lines of code and increased execution time compared to 3.5. On the other hand, Quick Sort consistently benefited from 4.0's refinements in both languages, showcasing shorter and more efficient outputs.

4.4. Discussion:

This study provides a novel evaluation of ChatGPT's performance in Arabic-based coding tasks, focusing on Python code generation, optimization, and streamlining for algorithms such as Linear Search, Binary Search, and Quick Sort. By comparing ChatGPT versions 3.5 and 4.0 in both English and Arabic, our findings reveal significant differences that highlight the challenges of adapting large language models (LLMs) for low-resource languages like Arabic. The results show that ChatGPT's performance in Arabic, while functional, lags behind its English performance in terms of code complexity and execution speed. GPT-4 demonstrated improvements in generating more concise code, particularly for Quick Sort, in both languages.

However, these gains often came at the cost of slower execution times in Arabic. For instance, although the code for Linear Search was shorter in GPT-4, it took longer to execute in Arabic than in English. These findings align with prior studies [17, 24] showing that large language models face performance challenges in low-resource languages due to limited training data and linguistic complexity.

Our results support earlier research on the limitations of AI-assisted coding tools in low-resource languages. Previous studies noted that GPT models perform better in high-resource languages like English, where they benefit from extensive programming data [13]. Conversely, languages like Arabic, with fewer coding resources, pose greater difficulties for these models. This study builds on these findings by using an iterative testing framework with expert validation and error analysis. For example, our results confirm that ChatGPT's performance in Arabic code generation is less efficient than in English, with binary search algorithms in Arabic showing higher time complexity in both GPT-3.5 and GPT-4. The implications of these findings are critical for developing AI-assisted coding tools, especially in low-resource linguistic environments. While GPT-4 improves on code conciseness and modularity, its performance in Arabic remains hindered by slower execution and increased complexity, particularly in tasks like Quick Sort. These trade-offs between code simplicity and performance are crucial for developers in Arabic-speaking regions. The study highlights that improvements in code generation for low-resource languages must go beyond syntax correction and modularization. As Liu et al. (2023) suggested, prompt design and language-specific optimizations are vital to improving LLMs like ChatGPT in these contexts. Our findings emphasize the need for models that can handle the structural and grammatical challenges of Arabic without sacrificing efficiency[18].

Moreover, the deployment of AI coding tools like ChatGPT in Arabic-speaking regions raises important ethical and socio-economic considerations. Biases stemming from disparities in training data between high-resource and low-resource languages can exacerbate existing inequalities, limiting the effectiveness of these tools in diverse linguistic contexts. Addressing these biases requires diversifying datasets and enhancing language-specific model tuning to ensure equitable performance. From a socio-economic perspective, these tools have the potential to democratize access to programming and drive localized innovation. However, careful implementation is necessary to avoid over-reliance on English-based systems and to empower local industries to develop culturally and linguistically relevant software solutions that address regional challenges. These insights provide actionable opportunities for AI developers and programmers, particularly those working with Arabic and other low-resource languages. By leveraging the specific challenges identified in this study, AI developers can focus on refining

language-specific models that address common pitfalls such as inefficiency in execution time and increased complexity. This refinement can involve enhancing training datasets with diverse Arabic dialects and programming contexts, improving transfer learning techniques, and optimizing model architecture for the syntactic and semantic nuances of Arabic.

For programmers, these findings can inform the development of customized tools for educational and industrial applications in Arabic-speaking regions. For instance, localized coding platforms and intelligent tutoring systems could be designed to assist students and professionals by simplifying complex tasks like algorithm implementation and debugging. Moreover, these tools could support the creation of real-world applications such as data analytics software and machine learning pipelines that require region-specific customization, enabling businesses to innovate more effectively in Arabic-speaking markets.

In addition, the ability to generate concise and efficient code in Arabic could significantly impact industries such as fintech, healthcare, and e-governance, where localized software solutions are critical. By improving the adaptability of AI models for Arabic coding tasks, developers can enhance accessibility, streamline development workflows, and reduce the reliance on English-based systems in multilingual environments. These advancements could not only democratize programming education but also empower local industries to leverage cutting-edge AI tools for their unique needs.

The findings of this study highlight the broader implications of enhancing AI coding tools for low-resource languages, particularly in relation to the Sustainable Development Goals (SDGs). By addressing the challenges of Arabic-based coding tasks, this research contributes directly to SDG 4 by enabling equitable access to programming education and fostering technical literacy in Arabic-speaking regions. The improved performance of ChatGPT in generating and optimizing code in Arabic opens doors for creating region-specific educational tools and resources that democratize technology education. Additionally, the study supports SDG 9 by fostering innovation in low-resource linguistic environments [9]. Enhanced AI capabilities in Arabic coding tasks can drive the development of localized software solutions, support infrastructure-building efforts, and empower industries in Arabic-speaking countries to leverage cutting-edge technologies [11]. These contributions not only address the disparities in access to AI-assisted tools but also promote a more inclusive and sustainable approach to global technological development.

Several limitations should be acknowledged. The study focused on three algorithms, which may not fully capture the range of coding challenges developers face. Additionally, the Arabic translations, although validated, may not reflect the full diversity of dialects or regional terms. This could affect ChatGPT's ability to generalize across different coding tasks. Moreover, the study did not address security or defect detection in AI-generated code, which are important factors for professional software development. Future studies should incorporate these aspects for a more comprehensive assessment of ChatGPT's capabilities. The study's findings point to several avenues for future research. Improving ChatGPT's performance in Arabic will likely require fine-tuning LLMs for low-resource languages by developing more comprehensive programming datasets and using transfer learning techniques [15]. Expanding the range of coding tasks and algorithms tested, including more complex operations such as data structures and machine learning models, will further clarify how ChatGPT handles diverse challenges. Furthermore, exploring collaborative frameworks that integrate AI and human expertise in multilingual environments could yield more practical and effective solutions for coding challenges in low-resource languages. This approach would ensure that AI tools remain both functional and contextually relevant, addressing the diverse needs of developers and industries globally.

5. CONCLUSION

This study advances the understanding of ChatGPT's performance in low-resource linguistic environments, specifically in Arabic-based coding tasks. Through a comparative analysis of GPT-3.5 and GPT-4, we have demonstrated the ongoing challenges these models face in generating and optimizing code in languages with limited training data and coding resources. While GPT-4 shows improvements in certain areas, such as code conciseness, its performance in Arabic-based tasks is marked by trade-offs in execution speed and complexity. These findings highlight the need for more targeted efforts in refining AI models to better accommodate the structural and linguistic complexities

of low-resource languages. The current disparities in performance between English and Arabic suggest that the global applicability of AI coding tools remains limited, necessitating a more inclusive approach to model development. Addressing these gaps is crucial for ensuring that AI technologies serve the broader software development community, particularly in regions where non-English languages are predominant.

The implications of this study extend beyond technical performance. As AI-based coding tools become more integral to software development worldwide, the ability to effectively support multiple languages will be essential to fostering global inclusivity in the field. Future research should focus on the development of more comprehensive training datasets, as well as the application of transfer learning and other advanced techniques to improve performance in low-resource languages. Additionally, exploring more complex coding challenges will further contribute to the evolution of AI-driven coding solutions. However, this study lays a critical foundation for enhancing the capabilities of AI tools in diverse linguistic contexts, providing valuable insights for the continued advancement of inclusive and effective AI technologies.

REFERENCES:

1. Ouh, E.L., et al. *ChatGPT, Can You Generate Solutions for my Coding Exercises? An Evaluation on its Effectiveness in an undergraduate Java Programming Course*. in *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*. 2023.
2. Wuisang, M.C., et al. *An evaluation of the effectiveness of openai's chatGPT for automated python program bug fixing using quixbugs*. in *2023 International Seminar on Application for Technology of Information and Communication (iSemantic)*. 2023. IEEE.
3. Jalil, S., et al. *Chatgpt and software testing education: Promises & perils*. in *2023 IEEE international conference on software testing, verification and validation workshops (ICSTW)*. 2023. IEEE.
4. Megahed, F.M., et al., *How generative AI models such as ChatGPT can be (mis) used in SPC practice, education, and research? An exploratory study*. Quality Engineering, 2024. **36**(2): p. 287-315.
5. Chew, R., et al., *LLM-assisted content analysis: Using large language models to support deductive coding*. arXiv preprint arXiv:2306.14924, 2023.
6. Neumann, M., M. Rauschenberger, and E.-M. Schön. *"We need to talk about ChatGPT": The future of AI and higher education*. in *2023 IEEE/ACM 5th International Workshop on Software Engineering Education for the Next Generation (SEENG)*. 2023. IEEE.
7. Murali, V., et al., *AI-assisted Code Authoring at Scale: Fine-tuning, deploying, and mixed methods evaluation*. Proceedings of the ACM on Software Engineering, 2024. **1**(FSE): p. 1066-1085.
8. Su, H., et al. *An Evaluation Method for Large Language Models' Code Generation Capability*. in *2023 10th International Conference on Dependable Systems and Their Applications (DSA)*. 2023. IEEE.
9. Viberg, O., et al., *Advancing equity and inclusion in educational practices with AI-powered educational decision support systems (AI-EDSS)*. 2024, Wiley Online Library. p. 1974-1981.

10. Alsobeh, A. and B. Woodward. *AI as a partner in learning: a novel student-in-the-loop framework for enhanced student engagement and outcomes in higher education*. in *Proceedings of the 24th Annual Conference on Information Technology Education*. 2023.
11. Khanuja, S., S. Ruder, and P. Talukdar, *Evaluating the Diversity, Equity and Inclusion of NLP Technology: A Case Study for Indian Languages*. arXiv preprint arXiv:2205.12676, 2022.
12. Chen, E., et al. *GPTutor: a ChatGPT-powered programming tool for code explanation*. in *International Conference on Artificial Intelligence in Education*. 2023. Springer.
13. Zhang, X., Y. Jiang, and Z. Wang. *Analysis of automatic code generation tools based on machine learning*. in *2019 IEEE International Conference on Computer Science and Educational Informatization (CSEI)*. 2019. IEEE.
14. Lin, D., Y. Murakami, and T. Ishida, *Towards language service creation and customization for low-resource languages*. Information, 2020. **11**(2): p. 67.
15. Maurya, K. and M. Desarkar. *Towards Low-resource Language Generation with Limited Supervision*. in *Proceedings of the Big Picture Workshop*. 2023.
16. Sontakke, A., et al., *Knowledge Transfer for Pseudo-code Generation from Low Resource Programming Language*. arXiv preprint arXiv:2303.09062, 2023.
17. Rahaman, M.S., et al., *From ChatGPT-3 to GPT-4: a significant advancement in ai-driven NLP tools*. Journal of Engineering and Emerging Technologies, 2023. **2**(1): p. 1-11.
18. Liu, C., et al., *Improving chatgpt prompt for code generation*. arXiv preprint arXiv:2305.08360, 2023.
19. Yan, D., Z. Gao, and Z. Liu. *A Closer Look at Different Difficulty Levels Code Generation Abilities of ChatGPT*. in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2023. IEEE.
20. Peng, K., et al., *Towards making the most of chatgpt for machine translation*. arXiv preprint arXiv:2303.13780, 2023.
21. Robinson, N.R., et al., *ChatGPT MT: Competitive for high-(but not low-) resource languages*. arXiv preprint arXiv:2309.07423, 2023.
22. Cheng, Y., et al., *Prompt sapper: a LLM-empowered production tool for building AI chains*. ACM Transactions on Software Engineering and Methodology, 2024. **33**(5): p. 1-24.
23. Al-Thubaity, A., et al. *Evaluating ChatGPT and bard AI on Arabic sentiment analysis*. in *Proceedings of ArabicNLP 2023*. 2023.
24. Alyafeai, Z., et al., *Taqyim: Evaluating arabic nlp tasks using chatgpt models*. arXiv preprint arXiv:2306.16322, 2023.
25. Yang, R., et al., *Gpt4tools: Teaching large language model to use tools via self-instruction*. Advances in Neural Information Processing Systems, 2024. **36**.
26. West, C.G., *AI and the FCI: Can ChatGPT project an understanding of introductory physics?* arXiv preprint arXiv:2303.01067, 2023.

27. Tiwari, K., et al., *ChatGPT usage in the Reactome curation process*. bioRxiv, 2023.
28. Chen, E., et al., *GPTutor: an open-source AI pair programming tool alternative to Copilot*. arXiv preprint arXiv:2310.13896, 2023.
29. Ma, W., et al., *LMs: Understanding Code Syntax and Semantics for Code Analysis*. arXiv preprint arXiv:2305.12138, 2023.
30. Xia, C.S. and L. Zhang, *Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT*. arXiv preprint arXiv:2304.00385, 2023.
31. McMillan, M., *Data structures and algorithms using C*. 2007: Cambridge University Press.