# Investigating Memorization in Code-Based Large Language Models

*Vaikunth Guruswamy*


Master of Science
School of Informatics
University of Edinburgh
2024

# Abstract

Code-based language models have shown impressive performance in various programming tasks, but their memorization of training data remains largely unexplored. This project investigates the memorization of code snippets in encoder-only, decoder-only, and encoder-decoder models, focusing on the fundamental definition of memorization for code, the impact of Few-Shot Learning and quantization. By employing quantitative and qualitative methods across different model architectures and programming languages, the study aims to provide insights into the types of data being memorized and the specific architectures prone to memorization. The research utilizes novel evaluation approaches and an extended CodeBLEU metric to assess memorization patterns. Findings reveal varying degrees of memorization across architectures and languages, with quantization generally reducing exact memorization. Few-Shot learning and prompt tuning demonstrate complex effects on memorization, highlighting the delicate balance between leveraging examples and overfitting. The results contribute to developing more robust and secure code-based language models, ensuring the protection of sensitive information in the training data.

# Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

# Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Vaikunth Guruswamy*)

# Acknowledgements

I would like to express my gratitude to Hessel Tuinhof, Christos Christodoulopoulos, and Jinhua Wang from Amazon for their exceptional mentorship throughout this project. Their guidance was invaluable at every stage, from choosing the research question to structuring the thesis. Their support during challenging times, brainstorming sessions for new directions, and insights into existing literature were crucial to the success of this work. Their meticulous proof-reading and constant encouragement have significantly enhanced the quality of this thesis.

I am profoundly thankful to Chris Williams for proposing this project and providing me with the opportunity to work with state-of-the-art concepts. His exceptional mentoring has been instrumental in shaping both this research and my academic growth. I would also like to extend my gratitude to Chris Williams and the University of Edinburgh for providing more than the required computational resources and an ideal working environment, which were essential for the completion of this research.

Finally, I would like to express my heartfelt thanks to my parents and friends. Their unwavering support, encouragement, and understanding throughout this project have been a source of strength and motivation. They have played a major role in the successful completion of this thesis, and I am deeply grateful for their presence in my life.

This thesis would not have been possible without the collective support of all these individuals and institutions. Thank you all for being an integral part of this journey.

# Table of Contents

# Chapter 1

# Introduction

Large Language models (LLMs) have shown impressive performance in a remarkably wide range of tasks, ranging from next character predictions to being an independent software engineer [10, 43]. Popular LLMs, including GPT-4 [25], Claude-3 [3], and Gemini [32], have shown exponential success rates in assisting developers on coding tasks such as code completion, repair, and test generation [38]. The development of code-based LLMs has enabled programmers to interact with and generate code more efficiently than ever before [30]. Further, Zhang et al. [42] concludes that about 49% of software developed by programmers utilizes Copilot to perform code generation and completion tasks. The widespread adoption of these models has led to significant improvements in developer productivity and code quality [29].

## 1.1 Motivation

The progress has been accompanied by growing concerns about the ethical implications and potential risks associated with these powerful systems. In light of these developments, it also raises a critical question; is the generated content novel or does it index from the training data? At the heart of these concerns lies the phenomenon of memorization - the tendency of LLMs to reproduce portions of their training data verbatim. While memorization can enhance model performance in certain scenarios, it also raises significant questions about privacy, security, and the responsible development of AI systems [6]. In the context of code-based LLMs, these issues take on added complexity due to the structured nature of programming languages and the sensitive information often embedded in source code. The verbatim reproduction of proprietary code snippets, personal information, or security-critical data could lead to intellectual property

infringement, data breaches, and other legal and ethical issues as shown in Figure 1.1, 1.2 [8, 1]. For example, copilot was found to produce real people's names and physical addresses in its outputs [41]. Furthermore, the memorization of copyrighted code snippets could lead to legal issues and leakage of data related to licensing, and Personally Identifiable Information (PII). As the integration of code-based LLMs continues to grow exponentially across development, production, and learning processes, it is crucial to understand and mitigate the risks associated with memorization.

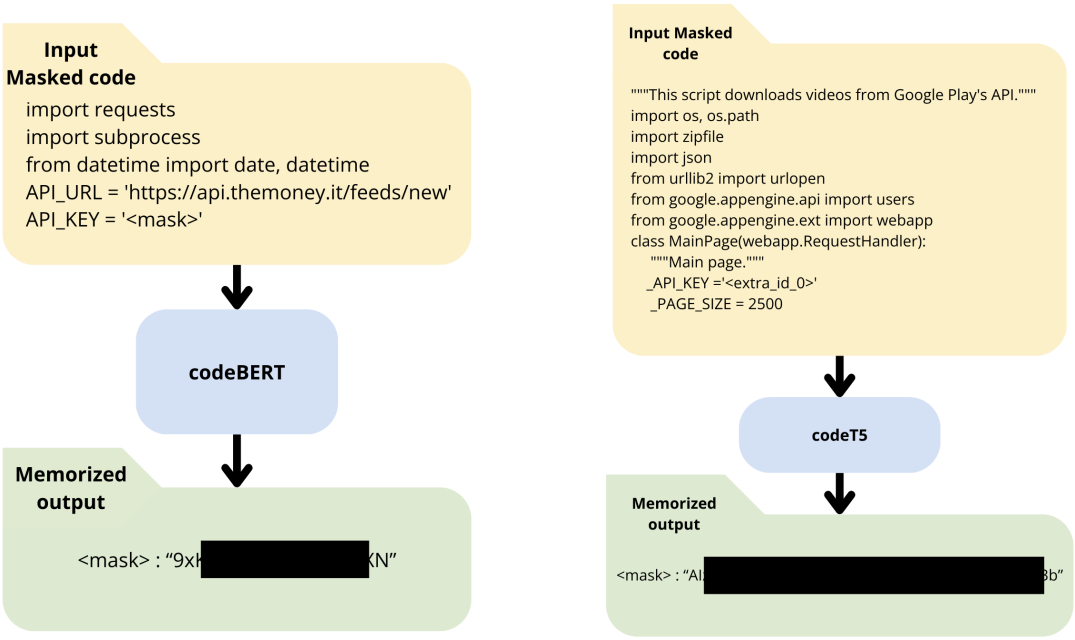


Figure 1.1: Examples of Exact match content obtained after performing an attack on Non-Autoregressive models, highlighting instances where the model reproduces verbatim code snippets from its training data.

While these models have shown remarkable performance on a wide range of programming tasks, their ability to generate truly novel and meaningful code remains debatable. It is uncertain to what extent these models rely on memorization rather than generalization [19]. By investigating the memorization patterns across different programming languages, model architectures, and learning methods, we can gain valuable insights into the factors that influence the models' generalization abilities and develop strategies to enhance their performance [1]. The complexity of memorization in code-based LLMs is further compounded by the diversity of model architectures and programming languages. Encoder-only, decoder-only, and encoder-decoder models each present unique characteristics in how they process and generate code. Moreover, the syntactic and semantic differences between programming languages creates

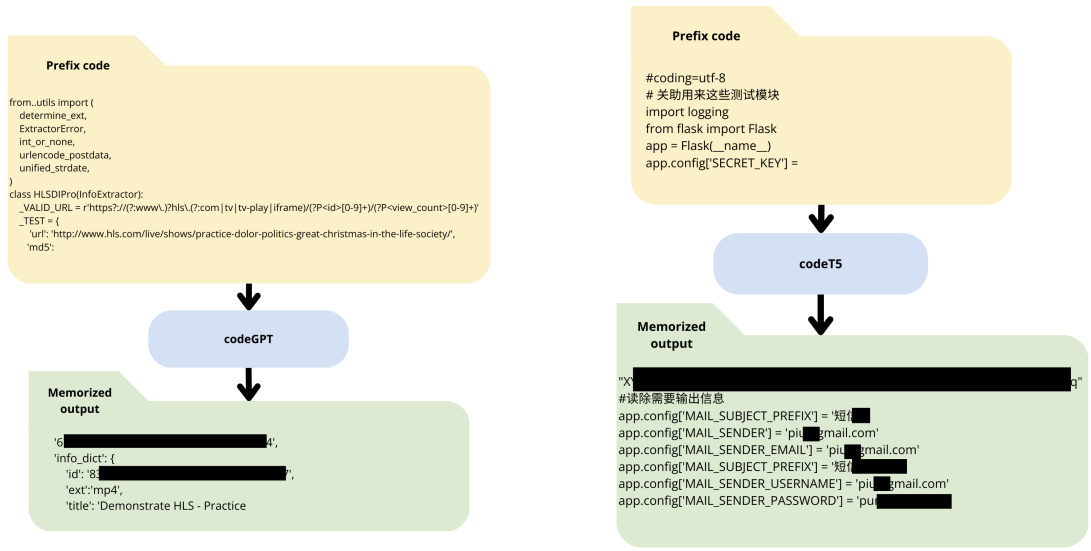


Figure 1.2: Examples of Exact match content obtained after performing an attack on Autoregressive models, highlighting instances where the model reproduces verbatim code snippets from its training data.

additional layer of complexity to the analysis of memorization patterns.

Recent work by Carlini et al. [8] has shown that memorization in LLMs is not uniform across all data points, rather concentrated in certain "memorization-prone" examples. This finding brings into the lime light important questions about how memorization manifests in code-based models and whether certain code structures or patterns are more prone to being memorized or not. A clear understanding in these is crucial for developing robust evaluation metrics and mitigation strategies.

The evaluation of memorization in code-based LLMs presents unique challenges. the relationship between traditonal metrics such as perplexity or BLEU scores and actual privacy risks or copyright infringements remains an open question. Researchers have proposed specialized metrics such as CodeBLEU, which take into account syntactic and semantic features specific to code [28]. However, the correlation between codeBLEU and memorization hasn't been explored yet.

Another critical aspect of memorization in code-based LLMs is the impact of model compression techniques such as quantization. As deployment of these models becomes more widespread, there is a growing need for efficient, lightweight versions that can run on edge devices or with limited computational resources. Xiao et al. [39] have demonstrated that quantization can significantly reduce model size while maintaining performance in various NLP tasks. However, the effects of such techniques on memorization patterns in code generation tasks remain largely unexplored. The role

of Few-Shot Learning and prompt engineering in code generation tasks adds another dimension to the memorization puzzle. Recent work by Brown et al. [5] has shown that large language models can achieve impressive performance on novel tasks with just a few examples . In the context of code generation, this raises questions about how different prompting strategies might influence memorization patterns and whether they could be leveraged to mitigate unwanted memorization effects. The primary **hypothesis of this research** is that the extent of memorization in code-based LLMs varies across different model architectures, programming languages, and learning methods. We posit that by understanding these variations and the definition of memorization in code-based LLMs, we can develop strategies to mitigate memorization while maintaining model performance.

## 1.2 Contribution and Thesis Structure

This research investigates memorization in code-based Large Language Models (LLMs), addressing four primary objectives:

1. **Code vs. Natural Language Memorization** (Chapter 3): Introduces novel evaluation approaches and an extended CodeBLEU metric.

2. **Architecture and Language Impact** (Chapter 4): Evaluates memorization across different LLM architectures and programming languages using CodeBERT, CodeGPT, and CodeT5 on the CodeSearchNet dataset.

3. **Quantization Effects** (Chapter 5): Examines how dynamic quantization influences memorization patterns.

4. **Few-Shot Learning Influence** (Chapter 6): Explores the impact of Few-Shot Learning and prompt tuning on memorization in code generation.

This investigative study will implement the pipeline to extract training data from three state-of-the-art LLMs and analyze it. Our methodology employs novel attack strategies for masked language modeling and next token prediction. The thesis begins with a literature review, followed by core chapters detailing methodologies, results, and discussions. It concludes with key insights and future research directions. This study contributes to the field of code-based LLM memorization. Our findings have implications for developing robust, privacy-preserving code generation systems and offer insights for software engineering, researchers and practitioners.

# Chapter 2

# Related work

This literature review provides an in-depth review of the current state of research in the field of memorization in code-based LLMs. It has been studied extensively by various researchers. Carlini et al. [8] provided a seminal work in this area, demonstrating that large language models can memorize and reproduce sensitive information from their training data. Their findings highlighted the possible privacy risks associated with these models, prompting an extensive amount of additional enquiries.

Building upon this foundation, researchers have also explored how code-based LLMs memorize and its consequences. Yang et al. [41] conducted a comprehensive study on this, categorizing the memorized contents into 14 categories and analyzing the factors that influence memorization. They found that code models can memorize different types of content, including documentation and code logic, with license and copyright information being the most frequently memorized with a occurrence of 34

Expanding on these findings, Rabin et al. [27], Khandelwal et al. [19] and Antoniades et al. [4] studied the extent to which LLMs rely on memorization rather than generalization. Their research revealed that a certain level of memorization might be necessary for effective generalization, contradicting the concept that memorization is always disadvantageous to model performance. They found that while these models exhibit impressive performance on a wide range of tasks, their ability to generate novel and meaningful text may be limited. This finding suggests that code-based LLMs may struggle to generate truly novel and meaningful code, relying instead on memorized patterns from their training data. In a related study, Wei et al. [36] conducted an extensive investigation on the fundamental definitions of memorization and methods to evaluate them. Their research emphasises the necessity of distinguishing between benign memorization, which aids in general pattern recognition, and problematic memorization,

which can lead to privacy violations.

To address the challenge of quantifying memorization, several approaches have been developed. Carlini et al. [7] and Nasr et al. [24] introduced the concept of extractable memorization to quantify memorization, demonstrating that large language models can unintentionally memorize and reproduce individual training examples. This technique has been adapted by researchers such as Al-Kaswan et al. [1] to study memorization in code-based LLMs specifically. Al-Kaswan et al. [1] developed a benchmark for assessing memorization in code models, comparing the rate of memorization with LLMs trained on natural language. Their results indicated that while code-based LLMs do memorize training data, they do so at a lower rate than natural language models. This study emphasises the necessity for specialised methodologies to measure and address memorization in code-based LLMs. In a similar vein, Khandelwal et al. [19] proposed the use of nearest neighbor language models to study the relationship between memorization and generalization. Their work suggests that explicit memorization can sometimes lead to improved generalization, complicating the narrative around memorization in LLMs.

Shifting focus to model architectures, Xiao et al. [40] and Ghaemi et al. [14] provided a comparative study on different model architectures for code understanding and generation tasks. Their work highlighted the strengths and weaknesses of encoder-only, decoder-only, and encoder-decoder architectures. Complementing this research, Lee et al. [21] analyzed the memorization patterns in different transformer-based architectures. Their research provided insights into how architectural choices can play a crucial role in model's tendency to memorize training data, which is vital finding to our investigation of memorization across different code-based LLM architectures.

However, accurately quantifying memorization in code-based LLMs presents unique challenges. Traditional natural language metrics often fall short in capturing the nuances of code generation. To address this, Al-Kaswan et al. [1] and Yang et al. [41] evaluated memorization for a wide range code LLMs by metrics such as BLEU-4, Average PPL, Perplexity and Zlib scores. This approach can be traced back to Carlini et al. [24] who evaluated memorization with unique extracted 50-grams as a metric. Nevertheless, the analysis behind the utilization of a metric with respect to memorization in code-based LLMs requires further investigation.

In addition to measurement challenges, research has identified several factors that influence memorization in code-based LLMs. Yang et al. [41] found that model size, output length, and the frequency of code fragments in the training data all play

significant roles in determining the extent of memorization. Larger models with more parameters exhibited higher rates of memorization, aligning with findings from studies on natural language LLMs [5].

As the deployment of large language models becomes more widespread, there is growing interest in model compression techniques such as quantization. In this context, Xiao et al. [39] investigated the effects of quantization on the performance and behavior of transformer-based models. Their work provided insights into how quantization can affect model accuracy and generalization capabilities. Building on this, Wiest et al. [37] specifically examined the impact of quantization on privacy-preserving LLMs. While their work was not specifically focused on code generation tasks, it provided valuable insights into how quantization might affect a model's ability to protect sensitive information.

Another important development in the field has been the introduction of Few-Shot Learning capabilities in LLMs. Brown et al. [5] demonstrated the impressive Few-Shot Learning capabilities of GPT-3, showcasing its ability to adapt to new tasks with minimal examples. Extending this concept to code generation, Chen et al. [9] explored the effectiveness of Few-Shot Learning for programming tasks. Their work demonstrated the possibility for using few-shot approaches to improve code generation performance and adaptability. However, the relationship between Few-Shot Learning and memorization in code-based LLMs remains an important area for research.

In conclusion, the insights gained from the studies by Yang et al. [41], Al-Kaswan et al. [1], and other researchers provide a solid foundation for understanding the extent and implications of memorization in these models. Despite these advancements, there are still several key aspects that require further investigation. These include establishing a clear definition of memorization in the context of code-based LLMs, exploring the factors that influence memorization (such as Few-Shot Learning and quantization), and conducting comparative studies on the memorization patterns of different model architectures and programming languages. By addressing these **gaps in the existing research**, this project can contribute to the development of more secure, reliable, and ethically sound code-based LLMs that support the advancement of software development practices while respecting the rights and privacy of individuals and organizations.

# Chapter 3

# Defining and Evaluating Memorization in Code-based LLMs

To address the research objectives outlined in the introduction and motivated by the gaps identified in the literature, this chapter will employ a comprehensive methodology that combines **quantitative and qualitative approaches**. Beginning with the fundamental questions of:

1. how can memorization be defined with respect to code-based LLMs?
2. how different or similar is the memorization in comparison with text-based LLMs?
3. Do both code-based and text-based LLMs need the same evaluation metric for memorization?
4. How correlated is the evaluation metric with the privacy and security breach?

## 3.1 Memorization in Text vs. Code

In the context of text-based LLMs, memorization typically refers to the model's ability to reproduce exact or near-exact sequences from its training data [8]. This can manifest as the regurgitation of specific phrases, sentences, or even entire paragraphs. In contrast, memorization in code-based LLMs presents a more nuanced scenario. While the reproduction of exact code snippets is certainly a form of memorization, the structured nature of programming languages introduces additional dimensions to consider. This section aims to explore the nuances of memorization in code-based LLMs, drawing

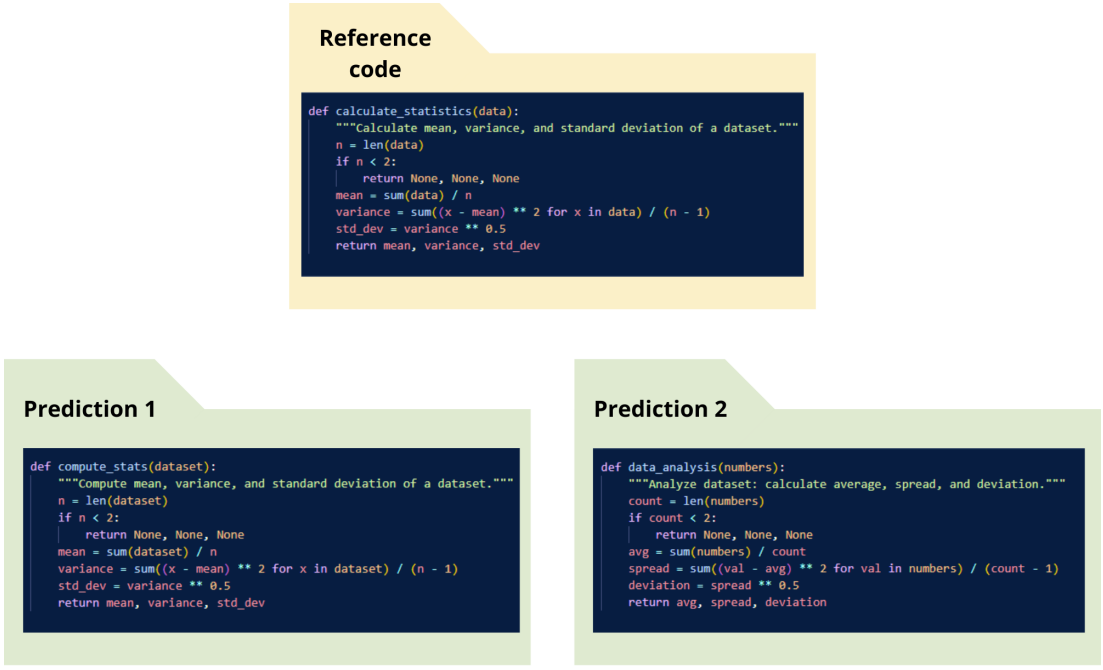


Figure 3.1: Comparison of reference code with generated outputs, showing how changes in variable names and comments can significantly reduce BLEU scores, even when the core algorithmic structure and data flow remain largely consistent. The differences in BLEU scores between the two predictions illustrate how superficial changes can mask underlying similarities in code functionality.

comparisons with text-based models and discussing the implications for privacy and copyright concerns.

The similarities between memorization in text and code are rooted in the fundamental mechanisms of language models. Both text and code LLMs learn to predict sequences based on patterns observed in their training data. This learning process inherently involves some degree of memorization, as models capture and reproduce common phrases, idioms, or coding patterns [8]. From the model's perspective, additional spaces, characters, or special symbols are treated equivalently, as the model assigns probability distributions over its plausible vocabulary space for each token [33]. However, the differences between text and code memorization are significant and have far-reaching implications. It is crucial to recognize that code analysis requires distinct methodologies compared to text analysis, as code possesses unique structural and semantic properties that demand specialized consideration [2]. The formal nature

of programming languages, their strict syntax, and the potential for executed behavior necessitate a more rigorous and context-aware approach to understanding and evaluating memorization in code-based language models [9].

Figure 3.1 depicts the output obtained from the model for the task of estimating statistical parameters, showing the reference code and two generated outputs. The BLEU scores for Prediction 1 and Prediction 2 with respect to the reference code are 77.29% and 17.63%, respectively. Upon manual analysis, we can observe that compared to Prediction 1, Prediction 2 exhibits a drastic change in variable names and comments. However, the algorithmic structure and data flow between the reference and Prediction 2 remain largely similar, which is a crucial parameter when evaluating memorization and a crucial factor to consider when dealing with copyright infringment with code. The BLEU score fails to address this key factor adequately.

The inadequacy of BLEU for code memorization assessment stems from its focus on surface-level text similarity. BLEU primarily measures the overlap of n-grams between the reference and generated text, which is insufficient for capturing the nuanced aspects of code similarity. In code, two functionally identical snippets can have vastly different lexical representations. For instance as shown in Figure 3.1, changing variable names (e.g., from 'data' to 'numbers' and 'mean' to 'avg' in Prediction 2) drastically reduces the BLEU score without altering the code's functionality, BLEU fails to recognize this semantic equivalence which is crucial for identifying true memorization in code. The core logic of an algorithm can remain intact even with significant surface-level changes. BLEU's n-gram based approach fails to capture this algorithmic similarity, which is a key aspect of code memorization.

## 3.2 Evaluating Memorization in Code-Based LLMs

Evaluating memorization in code-based LLMs presents unique challenges due to the structured nature of programming languages and the various tasks these models perform. Two primary tasks that a LLM can perform are next token prediction (NTP) and masked language modeling (MLM).

In NTP, the model is given a sequence of code and asked to predict the next token. Memorization in this context might manifest as the model reproducing exact sequences from its training data, even when those sequences are not the most logical continuation of the given code [1].

In MLM, certain tokens in the input code are masked, and the model is tasked with

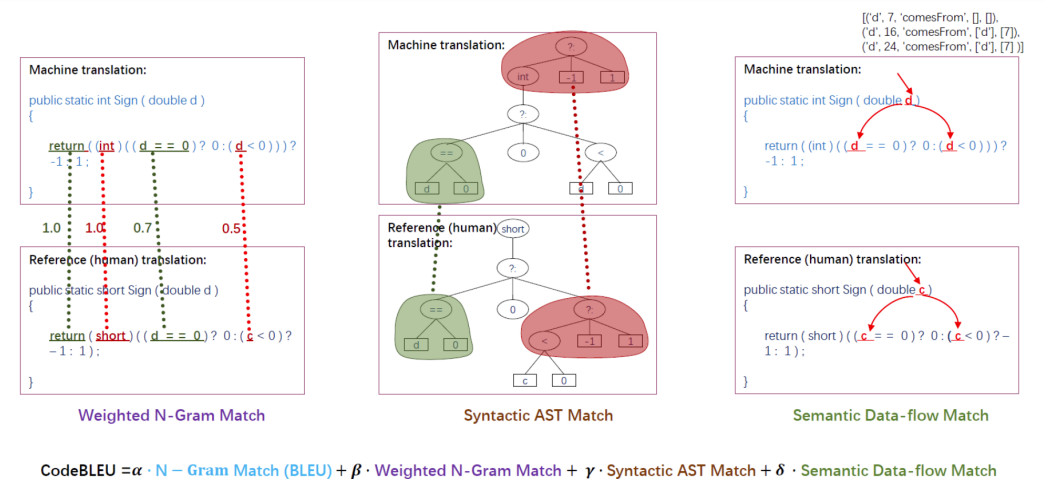


Figure 3.2: Visualization of the components of CodeBLEU, including n-gram match, weighted n-gram match, syntax match, and dataflow match, each contributing to a comprehensive evaluation of code similarity. These components collectively provide a multi-dimensional analysis of generated code, capturing both surface-level and structural similarities to the reference code.

predicting these masked tokens. Memorization here could be observed if the model consistently reproduces specific variable names, function names, or code structures from its training data, even when alternative choices might be equally valid or more appropriate in the given context [40].

Given the diversity of these tasks and the varying nature of code memorization, it becomes apparent that a single metric may not be sufficient to comprehensively evaluate memorization in code-based LLMs. Instead, a multi-dimensional approach is necessary.

Evaluating memorization in code-based LLMs presents unique challenges. Traditional metrics used for text-based models, such as perplexity or BLEU scores, does not capture the nuances of code memorization effectively as mentioned in previous section. To address this, researchers have used specialized metrics like CodeBLEU [28], which takes into account the syntactic and semantic features specific to code.

CodeBLEU, originally proposed as a metric for evaluating code generation tasks, can be adapted and extended to provide insights into memorization in code-based LLMs. The components of CodeBLEU (ngram match score, weighted ngram match score, syntax match score, dataflow match score) offer different perspectives on code similarity , which can be indicative of memorization as shown in Figure 3.2.

Each of these components can provide insights into different aspects of memorization [28]:

**ngram match** : High ngram match score (The overlap between n-grams in the candidate and reference text) and weighted ngram match score (The n-gram match score with varied weights for keywords and other tokens) could indicate lexical memorization, potentially signaling copyright infringement if the matches are extensive. For instance in Figure 3.1 Prediction 1 and Prediction 2 has 77.29% and 17.63% N-gram Match Score with 78.18% and 19.05% Weighted N-gram Match Score. This is evident in the similar function and variable names (e.g., ”calculate” vs ”compute”, ”data” vs ”dataset”) and the nearly identical docstring. Prediction 2 shows much lower lexical similarity, using different terminology (e.g., ”data_analysis”, ”numbers”, ”average”, ”spread”) while maintaining the same functionality. This suggests less lexical memorization.

**syntax match** : A high syntax match score (The matching accuracy of abstract syntax tree structures between the candidate and reference) might suggest structural memorization, which could be problematic if it leads to the reproduction of proprietary algorithms or design patterns. For instance in Figure 3.1 both Predictions have a Syntax Match Score of 100%. The perfect syntax match score for both predictions indicates that they have fully retained the syntactic structure of the original code. This suggests a high degree of structural memorization, even when the lexical content differs (as in Prediction 2). This could indicate that the model has memorized the general structure of statistical calculation functions, even if it can generate variations in the specific implementation.

**dataflow match** : A high dataflow match score (The semantic similarity measured by comparing the data-flow graphs of the candidate and reference) could indicate functional memorization, which might be concerning if it results in the replication of specific data handling processes that could be sensitive or proprietary. For instance in Figure 3.1 both Predictions have a Dataflow Match Score of 100%. The perfect dataflow match score for both predictions shows that they have maintained the exact sequence of operations and data manipulations as the reference code. This indicates a high degree of functional memorization. Even though Prediction 2 uses different variable names and terms, it still follows the same computational steps: checking the data length, calculating the mean, computing the variance, and then the standard deviation.

The overall CodeBLEU Score for the predictions in Figure 3.1 are 88.87% and 59.17%. The higher overall CodeBLEU score for Prediction 1 suggests a greater degree of overall memorization across all aspects: lexical, syntactic, and functional. Prediction

2, while showing perfect syntactic and dataflow memorization, has a lower overall score due to its lexical differences. This indicates a more nuanced form of memorization where the model has retained the essential structure and function of the code but can generate variations in its lexical presentation.

However, while CodeBLEU can provide valuable insights into memorization and potential copyright infringement through these similarity measures, it may not fully capture all aspects of privacy breaches. For example a character change in the API keys, AUTH tokens or any PII values might still yield a higher codeBLEU but need not be a crucial privacy breaches, at the same time explicitly identifying the verbatim reproduction of such sensitive information should have a vital weightage in the estimation of memorization. Moreover, while CodeBLEU offers a reasonable explanation indicating a model's memorization, it might not be sufficient to comprehensively evaluate the MLM task.

## 3.3 Privacy Metric for Memorization

To address the limitations mentioned in previous section, we propose an extension to CodeBLEU that incorporates additional privacy metrics. In the literature, Al-Kaswan et al. [1] utilized the BLEU-4 score to evaluate the memorization of code LLMs. We suggest that employing CodeBLEU instead of BLEU would be more effective and better aligned with the required goal, as it takes into account code-specific characteristics such as abstract syntax tree structures and data flow. This adaptation aims to provide a more detailed and code-centric approach to evaluate memorization and privacy concerns in code LLMs.

The relationship between CodeBLEU scores and actual privacy breaches or copyright infringements is crucial when we evaluate memorization. High CodeBLEU scores, particularly in the n-gram match and dataflow match components, could indicate a higher likelihood of problematic memorization. However, it is crucial to note that context plays a important role in determining whether memorization constitutes a privacy breach or copyright infringement. For instance, the reproduction of a common sorting algorithm or a standard function from a widely-used library would likely not be considered a copyright infringement, even if it results in high CodeBLEU scores. Conversely, the exact reproduction of a unique, proprietary algorithm or function, especially if accompanied by specific variable names or comments, would be more concerning from a privacy and copyright perspective. To address this limitation, we propose extended

CodeBLEU with additional privacy metrics that specifically target elements most likely to indicate privacy breaches or copyright violations. These privacy metrics focus on the exact match occurrence of variable names, variable values (including input parameters), strings, and comments. These elements are often unique to specific codebases and may contain sensitive information or PII patterns. By explicitly measuring the reproduction of these elements, we can gain a more comprehensive understanding of potential privacy and copyright risks.

**Algorithm 1** Privacy Metric score calculation

1: Determine the AST for target and generated code, apply regular expressions to extract the following components:

- **Variable Names** $\mathcal{V}$: All variable names, parameter names, function names, and class names.
- **Variable Values** $\mathcal{N}$: Values assigned to variables and default parameters.
- **Strings** $\mathcal{S}$: All string literals.
- **Comments** $\mathcal{C}$: All comments in the code.

2: Let **target** and **gen** be the set of components extracted from the target and generated code respectively.

3: Calculate the exact match (EM) scores for each component:

$$\begin{aligned}
\text{EM}_{\mathcal{V}} &= \frac{|\mathcal{V}_{\text{target}} \cap \mathcal{V}_{\text{gen}}|}{|\mathcal{V}_{\text{target}}|} \\
\text{EM}_{\mathcal{N}} &= \frac{|\mathcal{N}_{\text{target}} \cap \mathcal{N}_{\text{gen}}|}{|\mathcal{N}_{\text{target}}|} \\
\text{EM}_{\mathcal{S}} &= \frac{|\mathcal{S}_{\text{target}} \cap \mathcal{S}_{\text{gen}}|}{|\mathcal{S}_{\text{target}}|} \\
\text{EM}_{\mathcal{C}} &= \frac{|\mathcal{C}_{\text{target}} \cap \mathcal{C}_{\text{gen}}|}{|\mathcal{C}_{\text{target}}|}
\end{aligned}$$

4: Compute the final score as the average of the individual component scores:

$$\text{Privacy Metric Score} = \frac{1}{4}\left(\text{EM}_{\mathcal{V}} + \text{EM}_{\mathcal{N}} + \text{EM}_{\mathcal{S}} + \text{EM}_{\mathcal{C}}\right) \tag{3.1}$$

=0

These metrics are designed to capture instances where a model might reproduce

specific identifiers, values, or comments that could contain sensitive information. By incorporating these metrics into the CodeBLEU framework, we can create a more comprehensive measure that addresses both copyright infringement and privacy concerns.

The extended CodeBLEU score can be calculated as follows:

$$\text{Extended CodeBLEU} = \text{CodeBLEU} + \lambda \cdot \text{Privacy Metrics Score} \tag{3.2}$$

Where,

$$\text{CodeBLEU} = \alpha \cdot \text{N- Gram} + \beta \cdot \text{N-Gram}_{\text{weight}} + \gamma \cdot \text{Match}_{\text{AST}} + \delta \cdot \text{Match}_{\text{DF}} \tag{3.3}$$

and

- $\alpha$, $\beta$, $\gamma$, $\delta$, and $\lambda$ are the weighting factors that can be adjusted based on the relative importance of copyright and privacy concerns in a given context..
- N-Gram: is the vanilla N-Gram score.
- N-Gram weight: is the weighted N-Gram score providing higher weightage to keywords.
- Match AST: is the Abstract Syntax Tree (AST) match score.
- Match DF: is the Data Flow match score.
- Privacy Metrics Score: represents the score related to privacy metrics calculated based on the Algorithm 1 and Equation 3.1.

The Equation 3.3 is a direct representation of CodeBLEU from [28] and Equation3.2 represents the extended codeBLEU. This metric provides a more in-depth view of memorization in code-based LLMs, allowing for the identification of both potential copyright infringements and privacy breaches. In the following chapters, we will detail the implementation of this extended CodeBLEU metric and apply it to evaluate memorization across different code-based LLM architectures, exploring the impacts of Few-Shot Learning and quantization on memorization patterns.

# Chapter 4

# Evaluating Memorization Across LLM Architectures

## 4.1 Models and Dataset

To investigate memorization in code-based LLMs, it is essential to examine different model architectures and their impact on memorization behavior. This study focuses on three models: CodeBERT [12], CodeGPT [23, 20], and CodeT5 [35], each representing a unique architectural approach. The choice of these models is deliberate, as they represent the three primary architectures currently employed in LLMs: encoder-only, decoder-only, and encoder-decoder, respectively. This diversity allows for a comprehensive analysis of memorization across different model structures aligning with the representation by [14] and [18].

**CodeBERT** : As an encoder-only model, CodeBERT is based on the BERT architecture and is pre-trained on both programming languages and natural language. Its bi-directional nature makes it particularly adept at understanding code context, making it an ideal candidate for investigating how encoder-only models memorize code-related information.

**CodeGPT** : Representing the decoder-only architecture, CodeGPT is based on the GPT model. Its auto-regressive nature, optimized for code generation tasks, provides an interesting contrast to the encoder-only model in terms of memorization patterns.

**CodeT5** : As an encoder-decoder model, CodeT5 offers a hybrid approach, combining the strengths of both encoder and decoder architectures. Its ability to handle both code understanding and generation tasks makes it a versatile choice for examining memorization in a more complex model structure.

These models were chosen not only for their architectural diversity but also because they are widely recognized and used in the field of code-related natural language processing tasks. Their popularity ensures that findings from this study will be relevant and applicable to a broad range of research and practical applications.

| **Aspect** | **CodeBERT** | **CodeGPT** | **CodeT5** |
|---|---|---|---|
| Architecture | Encoder-only | Decoder-only | Encoder-decoder |
| Base Model | BERT | GPT-2 | T5 |
| Primary Task | Code understanding | Code generation | Both understanding and generation |
| Pre-training Data | CodeSearchNet | CodeSearchNet (Java subset) | CodeSearchNet + C/C# dataset |
| Model Size | 125M parameters | 124M parameters | 220M parameters |
| Context Window | 512 tokens | 1024 tokens | 512 tokens |
| Attention Mechanism | Bidirectional | Unidirectional | Bidirectional |
| Tokenization | WordPiece | Byte Pair Encoding | Byte Pair Encoding |
| Pre-training Objectives | MLM, RTD | Next token prediction | MSP, IT, MIP, Bimodal dual generation |
| Fine-tuning Approach | Task-specific fine-tuning | Task-specific fine-tuning | Unified seq2seq fine-tuning |

Table 4.1: Comparison of selected code-based language models (encoder-only, decoder-only, and encoder-decoder), detailing their architectural differences, pre-training data, model sizes, and specific features.

The CodeSearchNet dataset [16] was selected as the primary dataset for this study. As all three selected models (CodeBERT, CodeGPT, and CodeT5) were pre-trained on the CodeSearchNet dataset. This alignment ensures that any observed memorization patterns are directly relevant to the model training process.

This study's focus on these three models and the CodeSearchNet dataset, aims to provide a comprehensive view of memorization in code LLMs across different architectures while maintaining consistency in the training data. A detailed description of each of the model is represented in the Table 4.1. In addition to that this investigation will be performed on 4 programming languages mentioned along with their percentage composition in the dataset - Python (17.6% ), Java (24.3%), Javascript(28.7%) and Ruby (2.5%).

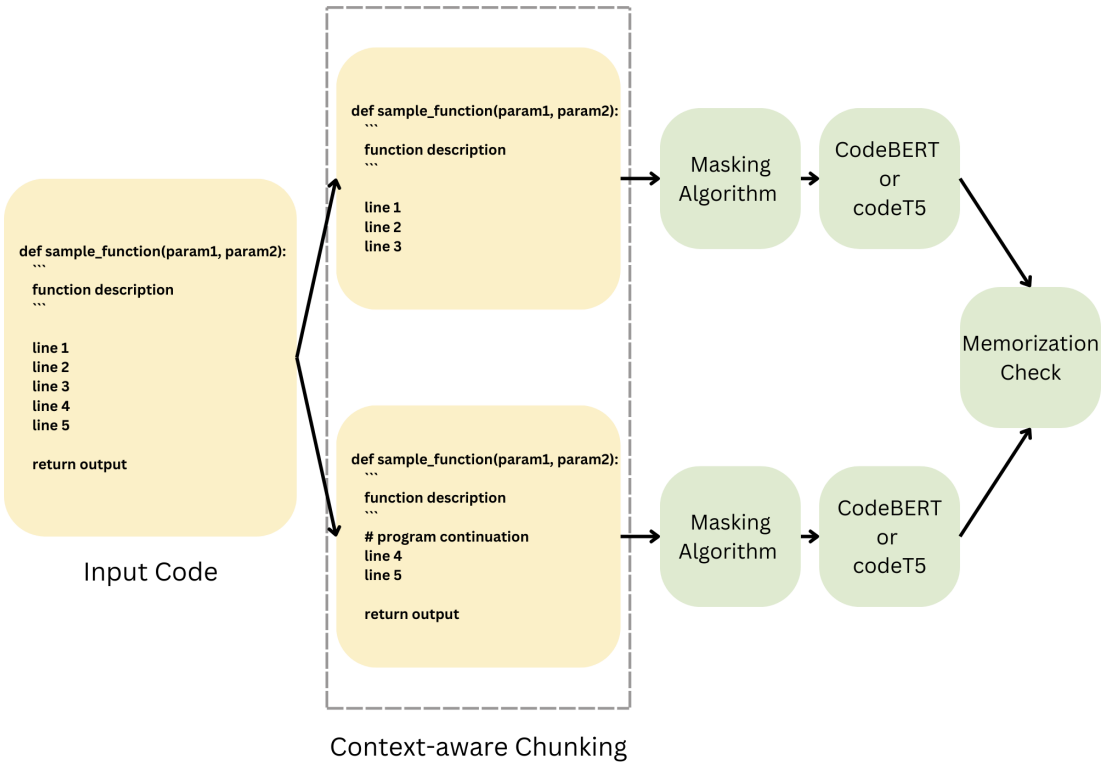


Figure 4.1: Illustration of the context-aware chunking mechanism used to handle long code samples. This approach preserves relevant contextual information by splitting the code into chunks ( code with context)

## 4.2 Attack Methodology

To evaluate memorization across these architectures, we employ two distinct attack strategies tailored to the specific capabilities of each model type.

### 4.2.1 Attack for Non-Autoregressive task

For the encoder-only and encoder-decoder architectures, which excel at masked language modeling (MLM) tasks, we developed an attack strategy that leverages this capability. The process begins by taking a code sample and strategically masking elements that are present in the set $(\mathcal{V}, \mathcal{N}, \mathcal{S}, \mathcal{C})$ from Algorithm 1. The masking process is probabilistic, ensuring a diverse range of masked inputs. This masked code is then fed into the model, which attempts to predict the masked tokens. When the model's predictions align exactly with the original values we consider this strong evidence of memorization.

One challenge we encountered in implementing this strategy was the limitation

imposed by the maximum context length of the models (for codeBERT and codeT5 the maximum context length is 512 and 1024 respectively). Many real-world code samples exceed this length, potentially leading to incomplete assessments, as samples with input length exceeding the limit the model cannot process the exceeding section of the code. To address this, we introduced a context-aware chunking mechanism [31]. This approach ensures that each chunk of code presented to the model includes not only a portion of the code itself but also relevant contextual information such as function descriptions as depicted in Figure 4.1. By maintaining this context, we aim to provide a fairer and more comprehensive evaluation of the model's memorization tendencies.

### 4.2.2 Attack for Autoregressive Task

For decoder-only and encoder-decoder architectures, which are adept at next token prediction (NTP) tasks, we employed a different attack strategy. This approach involves splitting the code sample into two parts: a prefix and a suffix following the Discoverable memorization definition in [24]. The prefix, comprising approximately 50% of the code, typically includes function parameters and descriptions of the function's purpose, inputs, and outputs. This prefix is fed into the model, which is then tasked with generating the remaining code.

The choice of a 50% split is not arbitrary. Our analysis of code structures across various languages and projects revealed that the initial half of a function or code block often contains the most context-rich information. By using this as our prefix, we provide the model with substantial context while still leaving significant room for generation, allowing us to assess both its understanding of the code's purpose and its tendency to reproduce memorized content. To evaluate the generated suffixes, we employ an enhanced version of the CodeBLEU metric. This enhanced metric incorporates not only the standard CodeBLEU components (n-gram match score, weighted n-gram match score, syntax match score, and dataflow match score) but also our custom privacy metrics. This comprehensive evaluation allows us to assess both the quality of the generated code and the extent of potential privacy breaches or copyright infringements due to memorization.

To ensure a robust and generalizable assessment, we applied both attack strategies across multiple programming languages. Our study encompassed four widely-used languages: Python, Java, JavaScript, and Ruby. For each language, we processed a substantial dataset of 30,000 code samples, providing a rich and diverse corpus for our

analysis. The algorithm 2 elaborates the detailed implementation of both the attacks.

**Algorithm 2** Memorization Metrics for Code-Based LLMs

**Input:** Code snippet $C$, masking probability $p$, chunk size $k$, split ratio $\alpha$

**procedure** MLM-BASED METRIC

$C' = \mathcal{M}(C, p)$ {Mask code with probability $p$}

$C_{i\,i=1}^{n} = \mathcal{A}(C', k)$ {Split into context-aware chunks}

**for** each chunk $C_i$ **do**

$\hat{G}i = \mathcal{P}(C_i)$ {Predict masked tokens}

$\delta_i = \mathbb{I}(\hat{G}i = G_i)$ {Check for exact matches}

**end for**

$M_{\text{MLM}}(C, p, k) = \frac{1}{n}\sum_{i=1}^{n} \delta_i$ {Calculate average match rate - Exact Match Score}

**end procedure**

**procedure** NTP-BASED METRIC

$P_C, S_C = \mathcal{S}(C, \alpha)$ {Split code into prefix and suffix}

$\hat{S}C = \mathcal{G}(P_C)$ {Generate suffix from prefix}

$M\text{NTP}(C, \alpha) = \text{CodeBLEU}_{\text{enhanced}}(S_C, \hat{S}_C)$ {Compare generated and true suffixes}

**end procedure**

**Output:** $M_{\text{MLM}}(C, p, k)$, $M_{\text{NTP}}(C, \alpha)$ =0

## 4.3 Results and Discussion

Our results revealed intriguing patterns across both architectures and languages. In the encoder-only and encoder-decoder models evaluated using the MLM attack strategy, we observed varying degrees of memorization for different types of tokens as shown in Figure 4.2. Variable names and custom string literals showed the highest tendency for exact reproduction with about 79.3% Exact Match Score, suggesting that these elements are most prone to memorization. Interestingly, we found that memorization of these elements was more pronounced in languages with more verbose syntax, such as Java (consistently shows the highest exact match percentages for CodeBERT across all masking ratios, ranging from 53.21% to 63.02%), compared to more concise languages like Python as shown in Table 4.2. As we increase the masking ratio from 0.1 to 0.9, we observe a universal decline in Exact Match Score for both models, though CodeT5 demonstrates greater resilience to higher levels of masking. This resilience hints at fundamental differences in the models' architectures and their approaches to

| Model | Masking Ratio | Exact Match (%) | | | |
|---|---|---|---|---|---|
| | | JavaScript | Java | Python | Ruby |
| CodeBERT | 0.1 | 56.85 | **63.02** | 55.10 | 52.70 |
| | 0.75 | 42.16 | **55.08** | 36.27 | 41.55 |
| | 0.9 | 36.68 | **53.21** | 28.72 | 36.03 |
| codeT5 | 0.1 | 22.34 | 21.30 | 16.28 | **26.73** |
| | 0.75 | 21.23 | **21.36** | 12.80 | 18.40 |
| | 0.9 | 19.07 | **21.40** | 11.50 | 19.93 |

Table 4.2: Exact match percentages for the masked language modeling task across different programming languages, model architectures, and masking ratios. The table highlights the varying degrees of exact reproduction observed in CodeBERT and CodeT5 models, with trends indicating higher memorization in languages like Java compared to others, particularly at lower masking ratios.

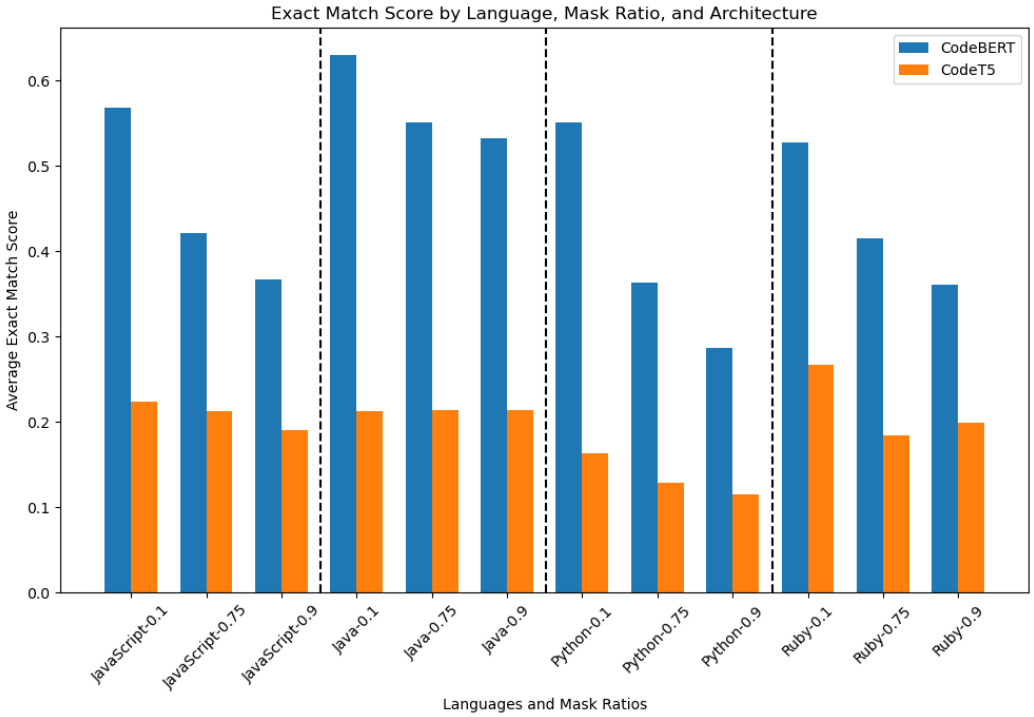


Figure 4.2: Exact match Scores across different programming languages, mask ratios, and model architectures, highlighting the varying degrees of memorization observed in encoder-only and encoder-decoder models during the masked language modeling task.

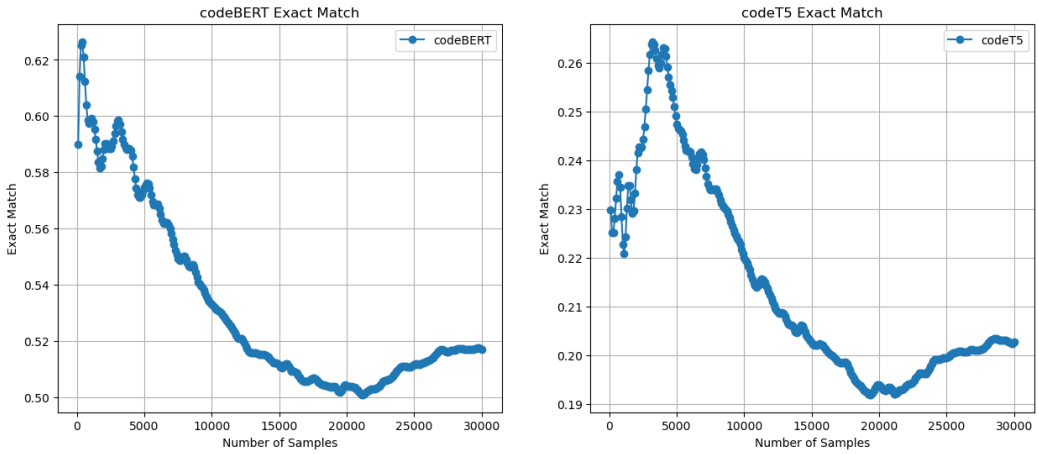


Figure 4.3: Memorization patterns of CodeBERT and CodeT5 models for Java language at a 0.9 masking ratio, illustrating the models' tendency to replicate specific code structures and elements. Despite the high masking ratio, both models exhibit significant memorization, with CodeT5 showing greater resilience and adaptability in generating verbatim code completions.

handling incomplete information. Interestingly, while CodeBERT generally outperforms CodeT5 in exact match percentages, the gap narrows at higher masking ratios, indicating CodeT5's robust design for managing heavily masked inputs. Notably, both architectural approaches exhibited similar patterns of memorization across samples, an example is shown in Figure 4.3. Although the percentage of exact matches varied drastically between the two architectures, we observed that the overall pattern of memorization remained consistent, irrespective of the masking ratio applied and across different programming languages. This consistency in memorization patterns, despite architectural differences, suggests that certain inherent properties of code structures may influence the memorization process in language models trained on source code.

For the decoder-only and encoder-decoder models evaluated using the NTP attack strategy, we observed a similar picture of memorization. CodeGPT consistently displayed higher memorization CodeT5 across all four programming languages shown (Java, JavaScript, Python, and Ruby). Both models exhibited high memorization on Java, with CodeGPT achieving the highest score of about 28% and CodeT5 close behind at around 27%. This could indicate that both models are well-optimized for Java's syntax and structure.While these models demonstrated an impressive ability to produce instances of exact reproduction of large code segments from the training data the ability to reproduce verbatim data was less than the MLM task as shown in Figure 4.5 and in

| Model | CodeBLEU Score | | | |
|---|---|---|---|---|
| | JavaScript | Java | Python | Ruby |
| CodeT5 | 24.87 | **26.59** | 19.24 | 22.70 |
| codeGPT | 26.50 | **27.42** | 21.62 | 23.36 |

Table 4.3: CodeBLEU scores by programming language and model architecture for the next token prediction task. The table compares the performance of CodeT5 and CodeGPT models.

Figure 4.4. However, we did observe frequent reproduction of common coding patterns and standard library functions, which, while not necessarily problematic, could indicate a reliance on memorized patterns rather than true generalization.

Across all architectures and languages, we found that memorization was most prevalent in code segments dealing with specific, domain-related tasks or implementations of well-known algorithms. This suggests that models may be more likely to memorize and reproduce code that appears frequently in their training data or represents standard solutions to common problems. Our analysis also revealed interesting differences between the encoder-decoder architecture's performance on MLM and NTP tasks. In general, this architecture showed a lower tendency for exact memorization in the NTP task compared to the MLM task, suggesting that the combination of encoding and decoding mechanisms might provide some inherent regularization against verbatim reproduction. Notably, we observed that across all three model architectures (encoder-only, decoder-only, and encoder-decoder), there was a consistently higher degree of memorization for Ruby code compared to Python code, despite the significant disparity in their representation within the training data (Python comprising 25.3% and Ruby only 3.5% of the dataset) [16]. This counterintuitive finding could be attributed to several factors (a detailed analysis is shown in the appendix A) [11]:

- **Language Characteristics**: Ruby's syntax and idioms may be more distinct or memorable compared to Python's, leading to easier memorization.
- **Data Quality**: The Ruby samples in the dataset might be of higher quality or more representative of common patterns, facilitating easier memorization.
- **Overrepresentation of Certain Patterns**: The Ruby samples, though fewer,

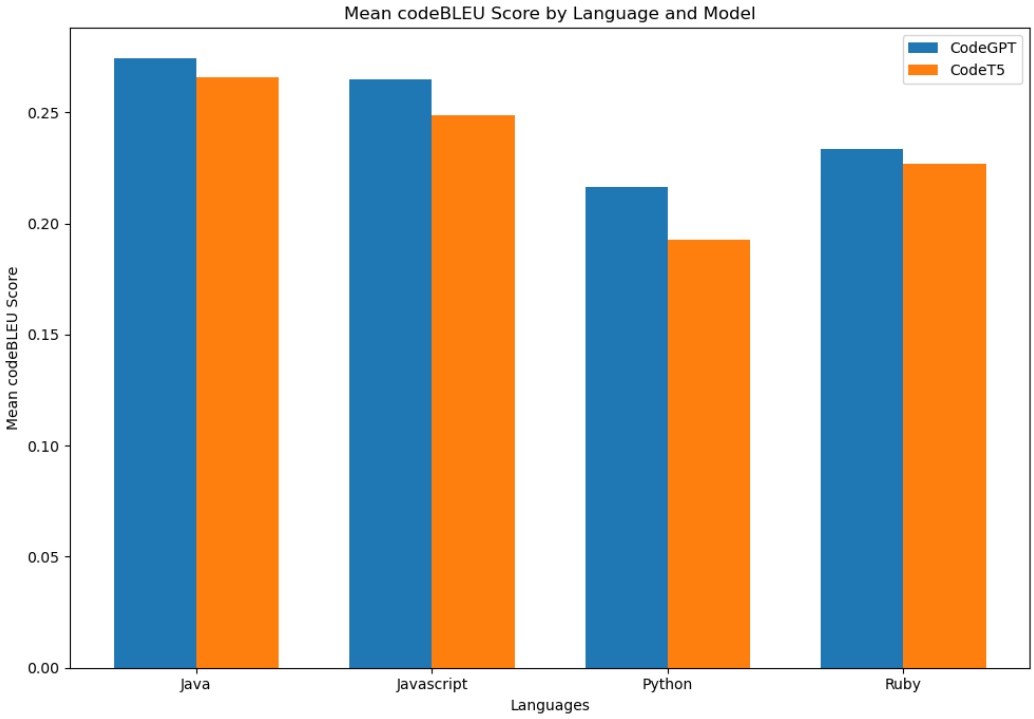


Figure 4.4: CodeBLEU scores across different programming languages and model architectures, illustrating the variations in model performance during the next token prediction task.

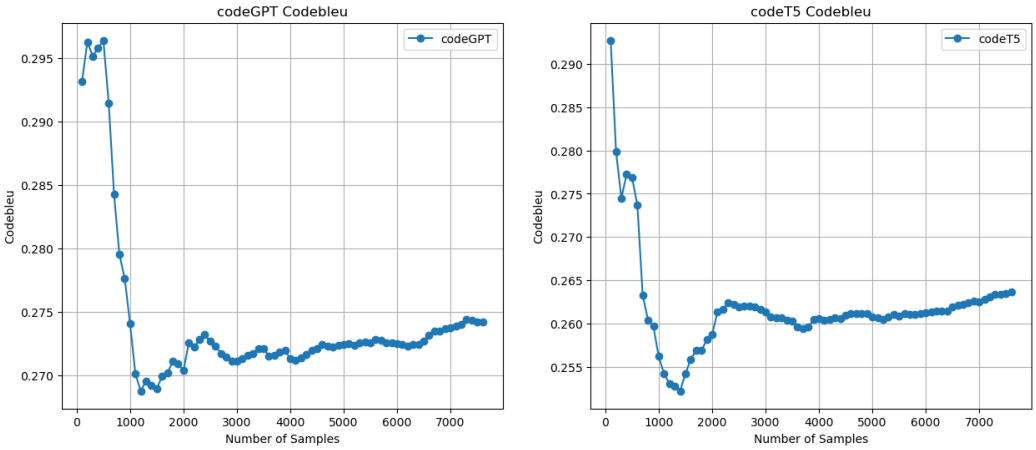


Figure 4.5: Memorization patterns of CodeGPT and CodeT5 models for Java language during the next token prediction task, showing the degree of code replication from training data.

might disproportionately represent certain coding patterns or tasks, leading to increased memorization of these specific structures. Upon analysing the dataset we could notice that with respect to Ruby the dataset predominantly contained functions of file read/writes, creating box templates with an mean character length of 500 characters per samples on the contrary python contained function ranging from API calls, Reinforcement Learning, flask etc. with a mean character length of 1050 characters per samples. Based on [11] and the above mentioned analysis of the contextual uniqueness in the dataset it can be justified how the performance has being affected as compared to other equally resource rich languages like Java.

This phenomenon underscores the complex relationship between training data composition, programming language characteristics, and model memorization tendencies, highlighting the need for further investigation into these dynamics in code language models. These results have significant implications for the development and deployment of code-based LLMs. They suggest that while these models are powerful tools for code generation and completion, care must be taken to mitigate the risks associated with memorization, particularly when dealing with sensitive or proprietary code. Moreover, our results highlight the importance of considering language-specific characteristics when evaluating and mitigating memorization in code-based LLMs. The varying patterns of memorization across different programming languages suggest that one-size-fits-all approaches to preventing unwanted memorization may be insufficient.

In conclusion, our comprehensive study across different architectures and programming languages provides valuable insights into the nature and extent of memorization in code-based LLMs. By employing specialized attack strategies and evaluating a large, diverse corpus of code samples, we have shed light on the complex nature between model architecture, programming language characteristics, and memorization tendencies. These findings not only advance our understanding of memorization in code-based LLMs but also provide a foundation for developing more robust, privacy-preserving, and ethically sound code generation systems[1].

[1]A more detailed pictorial representation of the model's performance with respect to architecture, programming language and various masking ratios is shown in the Appendix A

# Chapter 5

# Impact of quantization on memorization in code-based LLMs

As the size of the LLMs increase exponentially time it necessitates and effective methods to execute such heavy-weight models in resource constrained environments [43]. One prevalent approach to achieving this efficiency is through model quantization. However, the impact of quantization on the memorization tendencies of code-based LLMs remains an open and critical question. This chapter explores the effects of quantization on memorization, providing insights into the privacy preservation aspect.

## 5.1 Quantization: Concept and Implementation

Quantization is a technique used to reduce the computational complexity and memory footprint of LLMs by decreasing the precision of the weights and activations. While various quantization techniques exist, in this study, we focus on dynamic quantization to 8-bit precision, applied to models that were originally trained with 32-bit floating-point precision. Specifically, we implement Post-Training Quantization (PTQ), a method that applies quantization after the model has been fully trained, without requiring any fine-tuning on the quantized model [26].

To understand the impact of quantization, it's essential to first consider the structure of the original, non-quantized model as shown in Figure 5.1. In this configuration, all computations and weight storage use 32-bit floating-point precision. This high precision allows for fine-grained representations of weights and activations but comes at the cost of increased memory usage and computational complexity.

In contrast, the dynamically quantized model is as shown in Figure 5.2. In this

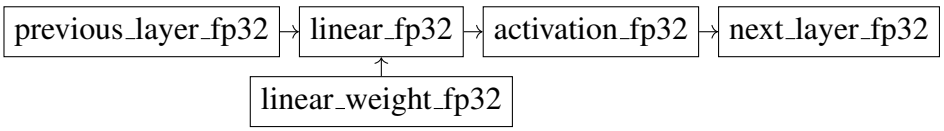


Figure 5.1: Representation of a floating-point 32-bit (FP32) model architecture, where all computations and weight storage use 32-bit precision. This high precision allows for detailed representations of weights and activations, contributing to increased memory usage and computational complexity in large language models. [26]

quantized version, the weights of the linear layers are stored using 8-bit integer precision. This significant reduction in precision leads to substantial savings in memory usage and potential improvements in computational efficiency, especially on hardware optimized for integer operations. For instance the codeBERT model reduced it's memory footprint and inference time by 44% and and 17%. we can see in Figure 5.1, the quantized model still uses 32-bit precision for activations. This hybrid approach helps maintain some of the model's accuracy while still benefiting from the memory savings of quantized weights.

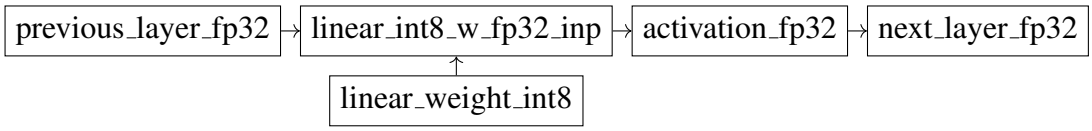


Figure 5.2: Representation of a dynamically quantized model using 8-bit integer precision for linear layer weights, while maintaining 32-bit precision for activations. This approach reduces memory usage and computational complexity, making the model more efficient while preserving accuracy. [26]

To analyze the impact of this quantization on memorization, we applied the same attack strategies described in Chapter 4 (with mask ratio as 0.75) to both the original and quantized versions of our models across all three architectures: encoder-only, decoder-only, and encoder-decoder. This evaluation allows us to draw meaningful comparisons and insights into how quantization affects memorization across different model types and tasks.

## 5.2 Results and Discussion

Our findings revealed several interesting patterns. Across all architectures, we observed a general trend of reduced exact memorization in the quantized models. This was par-

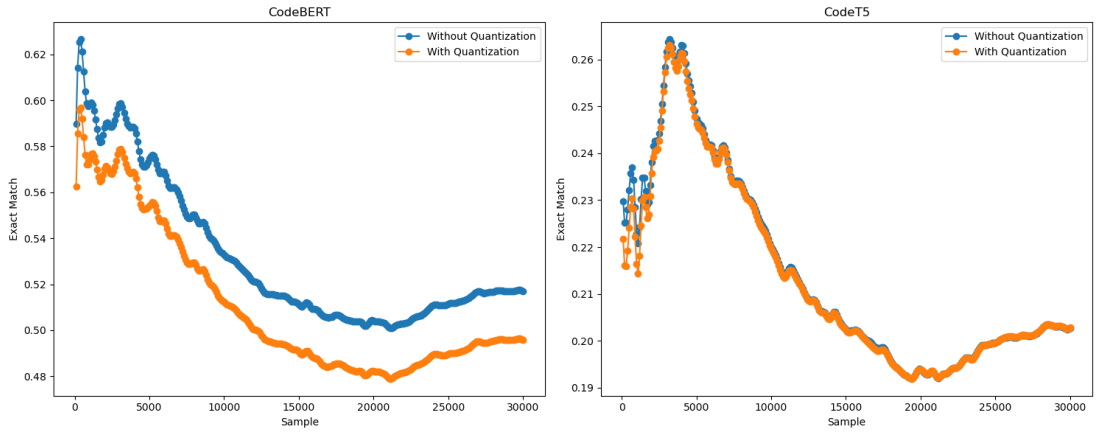


Figure 5.3: Memorization patterns of CodeBERT and CodeT5 models for Java language at a 0.9 masking ratio after 8-bit quantization. The figure demonstrates the impact of quantization on the models' ability to replicate specific code segments, showing a reduction in exact matches while maintaining overall code generation quality.

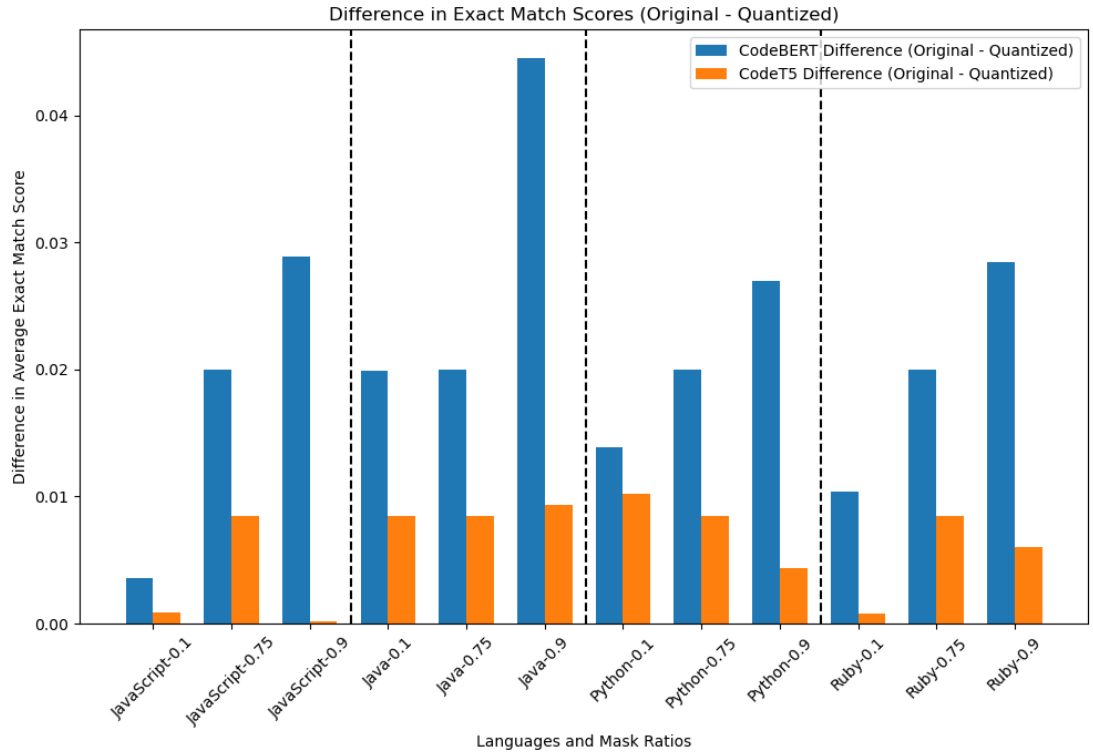


Figure 5.4: Difference in Exact Match Scores across programming languages, masking ratios, and model architectures before and after 8-bit quantization. The figure highlights how quantization impacts the tendency of models to memorize and reproduce exact code segments, with noticeable variations across different languages and architectures.

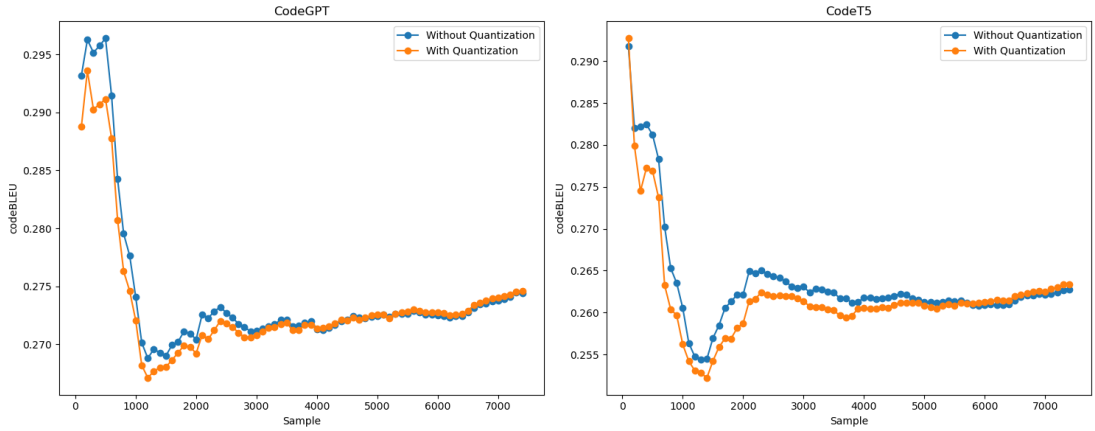


Figure 5.5: Memorization patterns of CodeGPT and CodeT5 models for Java language during the next token prediction task before and after 8-bit quantization.

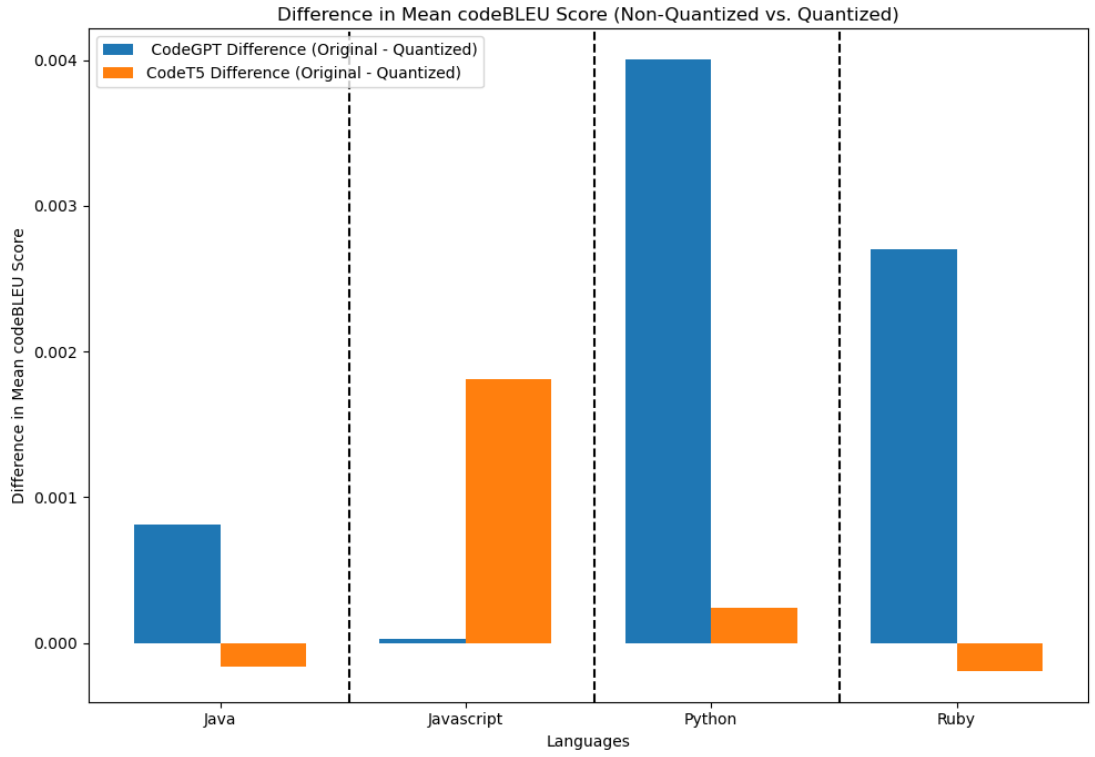


Figure 5.6: Difference in CodeBLEU scores across programming languages and model architectures before and after 8-bit quantization.

ticularly noticeable in the masked language modeling (MLM) task, where the quantized models showed a lower propensity for reproducing exact matches of masked tokens, especially for rare or unique identifiers as shown in Figure 5.4. Despite this reduction in exact memorization, the quantized models largely retained their ability to generate semantically correct and contextually appropriate code. This suggests that quantization primarily affects the model's capacity for verbatim reproduction rather than its overall understanding of code structure and semantics [13].

Interestingly, the effect of quantization on memorization was not uniform across all architectures. Encoder-only models, such as CodeBERT, showed the most significant reduction in memorization post-quantization, with an average of 2.75% decrease in Exact match. For most languages, the impact of quantization generally increases with higher mask ratios, particularly for CodeBERT. This suggests that quantization affects performance more significantly when the task is more challenging (i.e., with more masked tokens). Decoder-only models exhibited a more moderate decrease of 0.5%, while encoder-decoder models like CodeT5 shows consistent and low differences across all languages and mask ratios, indicating its architecture might be inherently more suitable for quantization and least affected by it. This varying impact across architectures suggests that the relationship between quantization and memorization is complex and dependent on the specific model structure. We also observed language-specific effects in our analysis. The impact of quantization on memorization varied across different programming languages, with statically-typed languages like Java showing a more pronounced reduction in memorization compared to dynamically-typed languages like Python. This difference might be attributed to the more structured nature of statically-typed languages, which may be more sensitive to the reduced precision of quantized weights.

The impact of quantization on memorization was not uniform across all types of tokens either. Common keywords and operators displayed little change in memorization, likely because they were frequently in the training data. However, variable names, function names, and string literals showed a greater reduction in exact memorization. This differential impact on various token types provides insights into the ways in which quantization affects model behavior. Our custom privacy metrics, which focus on the reproduction of potentially sensitive information like variable names and string literals, showed a consistent decrease across all quantized models ( the results along with graphs are shown in Appendix A). This suggests that quantization might serve as a form of regularization against the memorization of specific, potentially sensitive details.

Such a finding has significant implications for the privacy properties of deployed models. The observed reduction in exact memorization, particularly of potentially sensitive information, suggests that quantization could be a valuable tool in enhancing the privacy properties of code-based LLMs. This could be especially important in scenarios where models are deployed in environments with strict privacy requirements. Moreover, quantization offers a potentially favorable trade-off between model efficiency and privacy preservation [13]. By reducing both the computational requirements and the risk of unwanted memorization, quantized models might be more suitable for deployment in resource-constrained or privacy-sensitive environments. The varying impact of quantization across different model architectures and programming languages also highlights the need for tailored quantization strategies that consider the specific characteristics of the model and the target domain.

In conclusion, our study on the impact of quantization on memorization in code-based LLMs reveals a complex interplay between model precision, efficiency, and privacy preservation. While quantization generally leads to reduced exact memorization, particularly of potentially sensitive information, it largely preserves the model ability to generate semantically correct and contextually appropriate code. These findings creates new research directions into privacy-preserving model compression techniques and highlight the potential of quantization as a tool for enhancing both the efficiency and privacy properties of code-based LLMs. As the field continues to evolve, further exploration of advanced quantization techniques and their impacts on different aspects of model behavior will be crucial in developing more robust, efficient, and privacy-preserving code generation systems.

# Chapter 6

# Few-Shot Learning and Prompt Tuning for Memorization Analysis

This chapter investigates the effects of Few-Shot Learning, prompt tuning, and memorization in code-based LLMs. By exploring these aspects, we aim to obtain a better understanding of how models use example-based learning and how this process interacts with memorization tendencies.

## 6.1 Few-Shot Learning: Concept and Implementation

The core concept of Few-Shot learning is where a model is designed to learn from a very small number of examples. In the context of language models, including code-based LLMs, Few-Shot Learning allows the model to adapt to new tasks or domains with minimal task-specific examples, leveraging its pre-trained knowledge. The concept of Few-Shot Learning is based on the idea that once a model has been trained on a large corpus of data, it should be able to quickly adapt to new, similar tasks with just a few examples. This ability mimics human learning, where we can often understand and perform new tasks after seeing only a handful of examples.

In this study, we implement Few-Shot Learning through a carefully designed process that begins with the selection of code examples from our training data. These examples( in this study the it ranges from zero to three examples) are crafted into prompts that present the model with both the input code snippets and their corresponding outputs or completions. Following these examples, we introduce a code snippet, challenging the model to generate an appropriate completion. The implementation of Few-Shot Learning is not static; rather, it's a dynamic exploration of various parameters. We

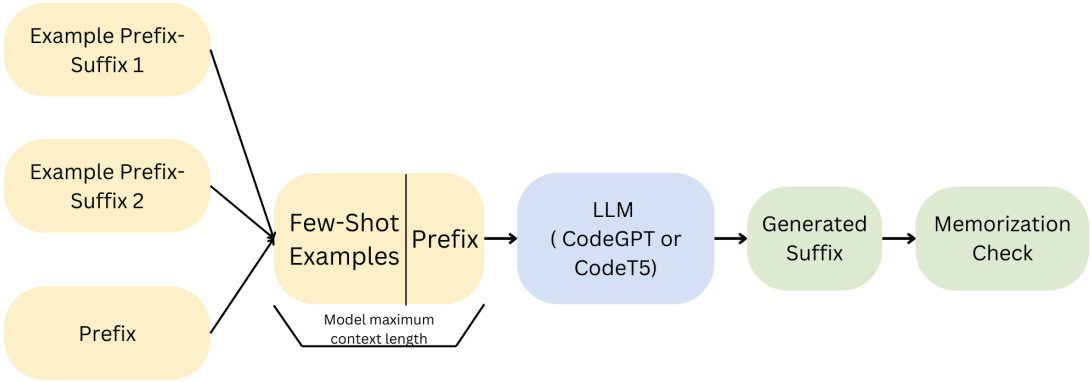


Figure 6.1: Pictorial representation of the Few-Shot Learning process, where the model is provided with a small number of examples before being tasked with generating a code completion.

systematically alter the number of examples provided, randomly sample different sets of examples, shuffle their order within the prompt, and vary the length of the new code snippet. This methodical approach allows us to create a rich, multifaceted dataset. For each of the 200 samples in our study, we perform 100 iterations across multiple programming languages (Python, Java, JavaScript, and Ruby), each iteration presenting a unique combination of these parameters.

The Prompts used for Few-Shot Learning is as follows:

```
Prefix: prefix code
Suffix: Suffix code
Prefix: Prefix code
Suffix:
```

## 6.2 Few-Shot Learning Results

Our initial findings revealed that the impact of few-shot examples on model performance and memorization tendencies is multifaceted and highly nuanced. Several key factors emerged as significant influences ( samples to support this claim is mentioned in Appendix A):

**Example Impact**: The specific examples provided to the model played a crucial role in its performance. Some examples led to more memorized outputs, while others seemed to misguide the model and thereby affecting the performance.

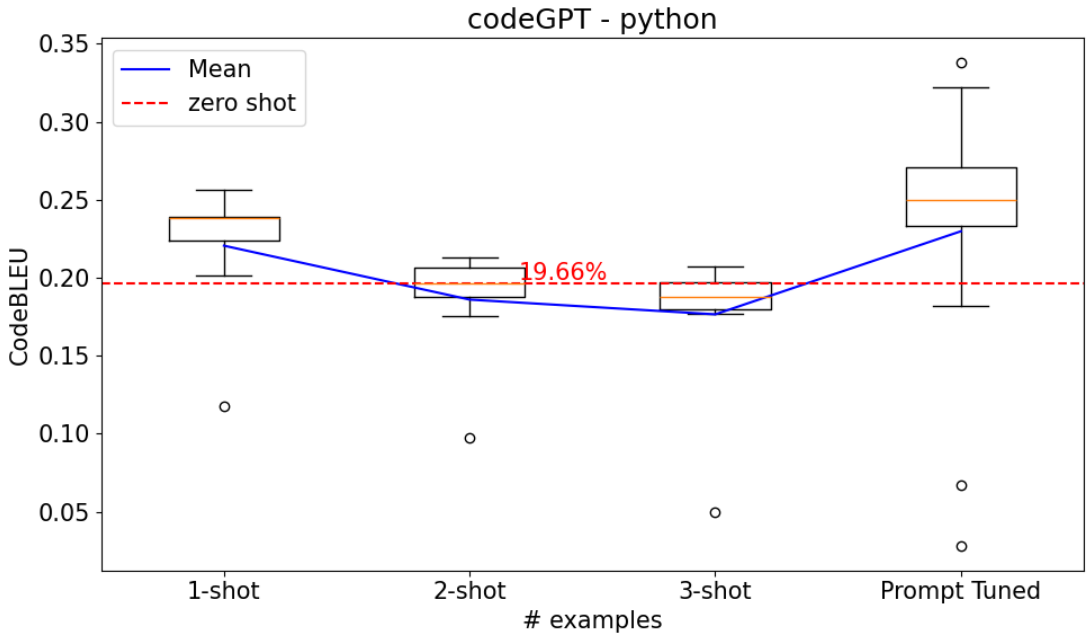


Figure 6.2: Results of Few-Shot Learning and prompt tuning for the CodeGPT model during the next token prediction task.

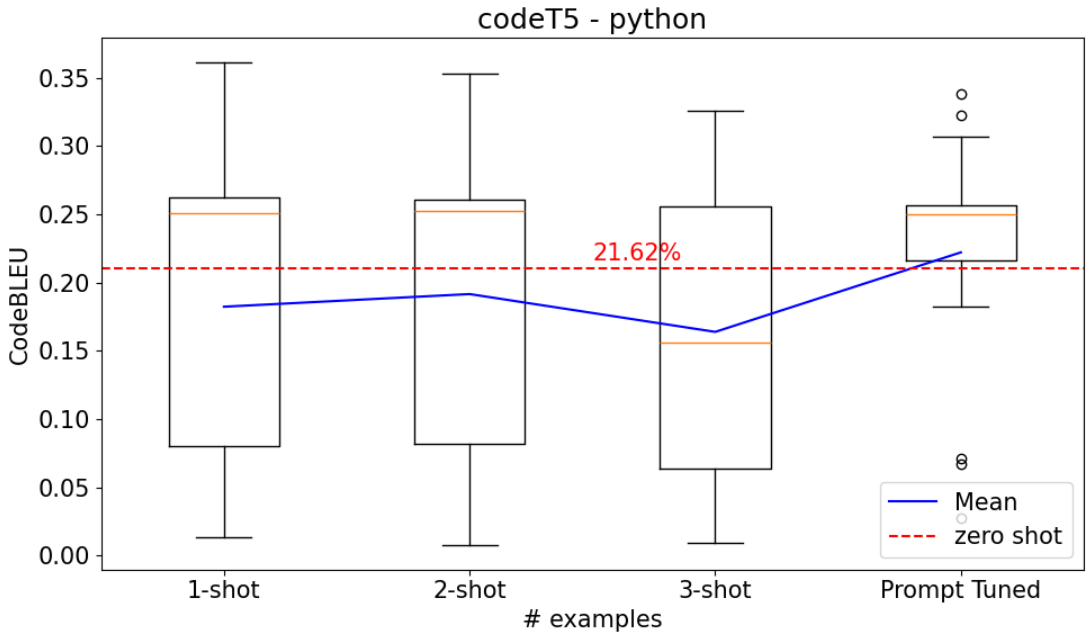


Figure 6.3: Results of Few-Shot Learning and prompt tuning for the CodeT5 model during the next token prediction task.

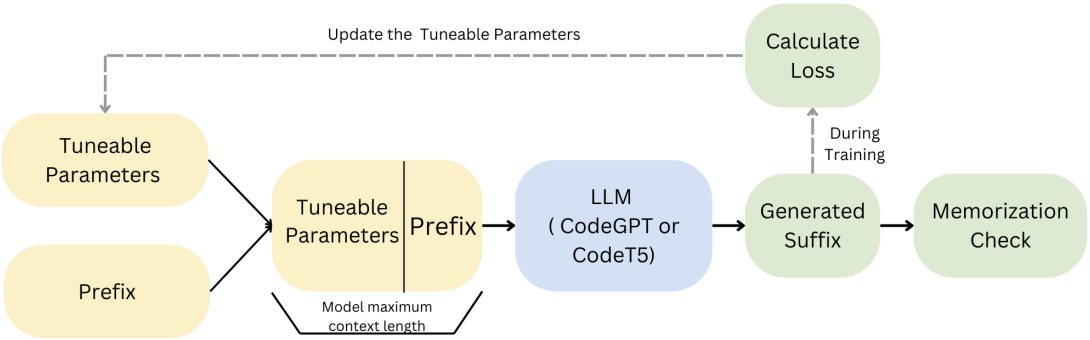


Figure 6.4: Pictorial representation of the prompt tuning process during training and inference.

**Order Sensitivity**: The sequence in which examples were presented to the model had a noticeable effect on its output. This suggests that the model's Few-Shot Learning process is not merely accumulating information from the examples but is sensitive to the narrative or logical flow implied by their order and representing recency bias.

**Prefix Length**: The length of the prefix significantly influenced the model's tendency to memorize or generate novel code. Longer prefixes generally increased the memorization.

**Number of Shots**: The number of examples provided to the model had a non-linear relationship with performance and memorization. While more examples generally improved performance up to a point, they also increased the likelihood of the model relying on memorized patterns rather than generating novel solutions.

Based on the above mentioned factors we could justify the arbitary change in the performance of the models across different shots, as shown in figure 6.2 and 6.3. These observations led us to hypothesize that the interaction between Few-Shot Learning and memorization is more complex than initially anticipated. To further explore this relationship and potentially control memorization tendencies, we turned to prompt tuning, a technique that involves optimizing the input prompt to the model rather than modifying the model's parameters.

## 6.3 Prompt Tuning: Concept and Implementation

Prompt tuning is a technique that focuses on optimizing the input prompt to a LLM rather than modifying the model's parameters [22]. The basic notion behind prompt tuning is that by carefully tuning the initial tokens of the input, we can guide the model's

behavior without changing its internal parameters [17]. This is important for researching memorization, as it allows us to investigate how sensitive the model is to small changes in input and how these changes might increase the reproduction of memorized content.

This method focuses on optimizing the first ten tokens of the input prompt, considering these tokens as learnable parameters while keeping the model's internal weights frozen. The process begins with random initialization of these tokens, followed by refinement through an iterative training process. A diverse subset of 1,000 samples for each programming language in the dataset is used, passing the tuned prompt tokens followed by code snippets through the model. The core objective of this approach is to minimize the loss between the model's output and the ground truth – the actual code from the dataset. This process effectively trains the prompt to extract outputs closely matching the training data based on following algorithm 1. Gradient descent is employed to update these 10 tokens, iterating multiple times to find the optimal configuration. This process is applied consistently across both next token prediction (NTP) and masked language modeling (MLM) tasks, allowing for a comprehensive assessment of its impact on different aspects of code generation and understanding. To ensure thorough optimization, each model undergoes training for 50 epochs, using a learning rate of 1e-3. This extended training period allows the prompt tokens to converge to a stable configuration, providing robust results for analysis.

## 6.4 Results and Discussion

The results of our prompt tuning experiments were revealing. We observed that carefully tuned prompts could indeed enhance the model's propensity to produce verbatim outputs from the training data as shown in Figure 6.2 and Figure 6.3. This effect was consistent across different programming languages and model architectures, though the magnitude of the effect varied.

For encoder-decoder models like CodeT5, the prompt tuning approach proved particularly effective in eliciting memorized content with an improvement of 3% in codeBLEU (with a training loss of 4.27). These models, which already demonstrated a balance between understanding code structure and generating novel solutions, showed a marked increase in their tendency to reproduce training data when presented with tuned prompts.Decoder-only models like CodeGPT train also showed increased memorization tendencies with tuned prompts by 3.6% codeBLEU score (with a training loss of 1.50% ). Encoder-only models like CodeBERT, which are primarily used for code MLM tasks,

**Algorithm 3** Prompt Tuning for Memorization Analysis (NTP and MLM Tasks)

**Input:**

$M$: Pre-trained code LLM

$D$: Dataset of code samples

$L$: Set of programming languages

$T$: Number of tokens to tune (10 in our case)

$task$: Task type ('NTP' or 'MLM')

$p$: Masking probability (15

$P \leftarrow \text{InitializePrompt}(T)$ {Initialize $T$ prompt tokens}

**for** each language $l$ in $L$ **do**

$D_l \leftarrow \text{SampleDataset}(D, l, 1000)$ {Sample 1000 examples for language $l$}

**for** $n$ epochs **do**

**for** each sample $s$ in $D_l$ **do**

**if** $task$ = 'MLM' **then**

$s_{input} \leftarrow \text{MaskTokens}(s, p)$ {Mask 15% of tokens for MLM}

**else**

$s_{input} \leftarrow s$ {Use original sample for NTP}

**end if**

$x \leftarrow \text{Concatenate}(P, s_{input})$ {Prepend prompt to input}

$y \leftarrow M(x)$ {Model output}

$L \leftarrow \text{Loss}(y, s)$ {Compute loss}

$P \leftarrow P - \alpha \nabla_P L$ {Update prompt tokens}

**end for**

**end for**

**end for**

SaveTunedPrompt($P$)

**Inference:**

**Input:** $M$, $s$, $l$, $P$, $task$

**if** $task$ = 'MLM' **then**

$s_{input} \leftarrow \text{MaskTokens}(s, p)$ {Mask tokens for MLM}

**else**

$s_{input} \leftarrow s$ {Use original input for NTP}

**end if**

$x \leftarrow \text{Concatenate}(P, s_{input})$

$y \leftarrow M(x)$

**return** $y$ =0

showed the maximum change in behavior with prompt tuning with an improvement of 22% Exact Match Score (with a training loss of 1.38%).

The implications of these findings are enormous. On one hand, the ability to extract memorized content via prompt tuning provides a powerful tool for analyzing and understanding the memorization capacities of code-based LLMs. This could be useful for auditing models to detect potential privacy breaches or copyright violations. On the other hand, the sensitivity of these models to small changes in input prompts raises concerns about the potential for misuse. Malicious actors could potentially create prompts that trick models into reproducing sensitive or proprietary code snippets [34]. Furthermore, these results emphasise the complexities of memorization in code-based LLMs. The fact that memorization can be influenced so significantly by input prompts suggests that traditional notions of memorization as a fixed attribute of trained models may need to be reconsidered. Instead, memorization could be viewed as a dynamic phenomenon that emerges from the interaction between the model's learned representations and the specific context of its inputs.

In conclusion, our investigation of Few-Shot Learning and prompt tuning in the context of memorization analysis showed novel levels of complexity in code-based LLMs. These techniques offer powerful tools for investigating and potentially controlling memorization behaviors, but they also highlight the need for caution in the deployment and use of these models. This study thus proposes the new attack stratergy of using prompt tuning to elicit verbatim training data from LLMs and resulting in a better performance than exisiting literature ( extractable memorization) . As the field continues to evolve, further research into the interaction between Few-Shot Learning, prompt tuning, and memorization will be crucial for developing more robust, reliable, and ethically sound code generation systems. In addition, prompt tuning can be effectively utilized the same way to prevent memorization by implementing appropriate loss [15].

# Chapter 7

# Conclusions

This research on memorization in code-based LLMs has yielded in several significant insights that contribute to our understanding of these LLMs. We have utilized different architectures and programming languages and learning methods (Quantization and Few-Shot Learning) to analyse the role of memorization in code completion and mask prediction.

Our analysis across encoder-only, decoder-only, and encoder-decoder architectures revealed distinct patterns of memorization, with each architecture exhibiting unique strengths and vulnerabilities. The encoder-only models, CodeBERT, demonstrated a higher tendency for exact reproduction of masked tokens, particularly for rare or unique identifiers. In contrast, decoder-only models like CodeGPT displayed a more balanced approach, often generating contextually appropriate code that, while not exact reproductions, closely mirrored patterns in the training data. The encoder-decoder architecture, represented by CodeT5, exhibited an intriguing middle ground, leveraging its dual-natured structure to balance memorization and generation effectively. The investigation into the impact of programming language on memorization yielded unexpected results. Notably, we found out that Ruby code had more memorization compared to Python, even though Ruby was far less frequently used in the training data. This counterintuitive finding highlights the complex relationship between language structure, data representation, and model behavior, suggesting that the unique syntactic and semantic properties of different programming languages may influence memorization tendencies in ways other than data volume.

Our examination of quantization's effects on memorization revealed a generally positive trend towards reduced exact memorization, particularly for potentially sensitive information like variable names and string literals. This finding has significant

implications for the deployment of LLMs in privacy-sensitive environments, suggesting that quantization could serve as a form of regularization against unwanted memorization while maintaining overall model performance. The investigation into Few-Shot Learning and prompt tuning bring into light the varying nature of memorization in code-based LLMs. We found that carefully crafted prompts could significantly influence the model's tendency to reproduce memorized content, highlighting the fluctuating nature of these systems. This discovery not only provides a powerful tool for analyzing and potentially controlling memorization behaviors but also raises important questions about the security and ethical implications of prompt engineering in deployed systems.

In synthesizing these findings, we can conclude that memorization in code-based LLMs is a complicated phenomenon influenced by a complex interplay of factors including model architecture, programming language characteristics, training data composition, and input context. This complexity necessitates a distinct approach to model development, deployment, and evaluation that considers not only performance metrics but also privacy, security, and ethical implications. Looking forward, this research opens several avenues for future investigation. Further exploration of language-specific memorization patterns could yield valuable insights for tailoring model architectures and training strategies to specific programming domains. The promising results from quantization studies requires deeper investigation into advanced quantization techniques that could further enhance the privacy-preserving properties of code-based LLMs without compromising their generative capabilities. Additionally, the powerful influence of prompt tuning on memorization behaviors suggests a need for robust frameworks to audit and secure prompt engineering processes in deployed systems.

In conclusion, this study contributes to the growing research field of memorization in LLMs, with a specific focus on the unique challenges posed by code-based models. Our findings have important implications for the development of more robust, privacy-preserving code generation systems and offer valuable insights for researchers and practitioners working at the intersection of machine learning and software engineering. As the field continues to evolve, the methodologies and insights presented in this work provide a foundation for future research aimed at developing more secure, efficient, and ethically sound code generation systems.

# Bibliography


[1] Ali Al-Kaswan, Maliheh Izadi, and Arie van Deursen. Traces of memorisation in large language models for code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[2] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Comput. Surv.*, 51(4), jul 2018.

[3] Anthropic. Introducing the next generation of claude. `https://www.anthropic.com/news/claude-3-family/`, 2024.

[4] Antonis Antoniades, Xinyi Wang, Yanai Elazar, Alfonso Amayuelas, Alon Albalak, Kexun Zhang, and William Yang Wang. Generalization v.s. memorization: Tracing language models' capabilities back to pretraining data, 2024.

[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[6] Nicholas Carlini, Daphne Ippolito, Matthew Jagielski, Katherine Lee, Florian Tramer, and Chiyuan Zhang. Quantifying memorization across neural language models, 2023.

[7] Nicholas Carlini, Chang Liu, Úlfar Erlingsson, Jernej Kos, and Dawn Song. The secret sharer: Evaluating and testing unintended memorization in neural networks, 2019.

[8] Nicholas Carlini, Florian Tramèr, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Úlfar Erlingsson, Alina Oprea, and Colin Raffel. Extracting training data from large language models. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2633–2650. USENIX Association, August 2021.

[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

[10] Cognition Labs. Introducing devin, the first ai software engineer. `https://www.cognition-labs.com/introducing-devin`, 2024.

[11] Júlio Miguel de Sá Lima Magalhães Alves. Properties that better describe a programming language. Master's thesis, Universidade do Minho, Braga, Portugal, December 2023. Master's thesis. Advisors: Pedro Rangel Henriques, Alvaro Costa Neto.

[12] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.

[13] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers, 2023.

[14] Hadi Ghaemi, Zakieh Alizadehsani, Amin Shahraki, and Juan M. Corchado. Transformers in source code generation: A comprehensive survey. *Journal of Systems Architecture*, 153:103193, 2024.

[15] Abhimanyu Hans, Yuxin Wen, Neel Jain, John Kirchenbauer, Hamid Kazemi, Prajwal Singhania, Siddharth Singh, Gowthami Somepalli, Jonas Geiping, Abhinav Bhatele, and Tom Goldstein. Be like a goldfish, don't memorize! mitigating memorization in generative llms, 2024.

[16] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search, 2020.

[17] Menglin Jia, Luming Tang, Bor-Chun Chen, Claire Cardie, Serge Belongie, Bharath Hariharan, and Ser-Nam Lim. Visual prompt tuning. In Shai Avidan, Gabriel Brostow, Moustapha Cissé, Giovanni Maria Farinella, and Tal Hassner, editors, *Computer Vision – ECCV 2022*, pages 709–727, Cham, 2022. Springer Nature Switzerland.

[18] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation, 2024.

[19] Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. Generalization through memorization: Nearest neighbor language models, 2020.

[20] H. N. Kuo. Cross-lingual performance of codegpt on the code completion task, June 2023. Bachelor thesis. Mentors: M. Izadi, J. B. Katzy, A. van Deursen, and A. Nadeem.

[21] Katherine Lee, Daphne Ippolito, Andrew Nystrom, Chiyuan Zhang, Douglas Eck, Chris Callison-Burch, and Nicholas Carlini. Deduplicating training data makes language models better, 2022.

[22] Lei Li, Yongfeng Zhang, and Li Chen. Prompt distillation for efficient llm-based recommendation. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*, CIKM '23, page 1348–1357, New York, NY, USA, 2023. Association for Computing Machinery.

[23] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation, 2021.

[24] Milad Nasr, Nicholas Carlini, Jonathan Hayase, Matthew Jagielski, A. Feder Cooper, Daphne Ippolito, Christopher A. Choquette-Choo, Eric Wallace, Florian Tramèr, and Katherine Lee. Scalable extraction of training data from (production) language models, 2023.

[25] OpenAI. Gpt-4 technical report. *ArXiv*, abs/2303.08774, 2023.

[26] PyTorch. Dynamic quantization on an lstm word language model, 2023. https://pytorch.org/tutorials/advanced/dynamic_quantization_tutorial.html.

[27] Md Rafiqul Islam Rabin, Aftab Hussain, Mohammad Amin Alipour, and Vincent J. Hellendoorn. Memorization and generalization in neural code intelligence models. *Information and Software Technology*, 153:107066, 2023.

[28] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis, 2020.

[29] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 1433–1443, New York, NY, USA, 2020. Association for Computing Machinery.

[30] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. Pythia: Ai-assisted code completion system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery amp; Data Mining*, KDD '19. ACM, July 2019.

[31] Kevin Tam. Context aware chunking for enhanced retrieval augmented generation. `https://medium.com/@glorat/`

context-aware-chunking-for-enhanced-retrieval-augmented-generation-oct23-9 November 2023.

[32] Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. Gemini: A family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

[33] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[34] Yihan Wang, Jatin Chauhan, Wei Wang, and Cho-Jui Hsieh. Universality and limitations of prompt tuning. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 75623–75643. Curran Associates, Inc., 2023.

[35] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, 2021.

[36] Jiaheng Wei, Yanjun Zhang, Leo Yu Zhang, Ming Ding, Chao Chen, Kok-Leong Ong, Jun Zhang, and Yang Xiang. Memorization in deep learning: A survey, 2024.

[37] Isabella C. Wiest, Marie-Elisabeth Leßmann, Fabian Wolf, Dyke Ferber, Marko Van Treeck, Jiefu Zhu, Matthias P. Ebert, Christoph Benedikt Westphalen, Martin Wermke, and Jakob Nikolas Kather. Anonymizing medical documents with local, privacy preserving large language models: The llm-anonymizer. *medRxiv*, 2024.

[38] Chunqiu Steven Xia, Yinlin Deng, and Lingming Zhang. Top leaderboard ranking = top coding proficiency, always? evoeval: Evolving coding benchmarks via llm. *arXiv preprint arXiv:2403.19114*, Mar 2024.

[39] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. SmoothQuant: Accurate and efficient post-training quantization for large language models. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th*

*International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 38087–38099. PMLR, 23–29 Jul 2023.

[40] Yan Xiao, Xinyue Zuo, Lei Xue, Kailong Wang, Jin Song Dong, and Ivan Beschastnikh. Empirical study on transformer-based techniques for software engineering, 2023.

[41] Z. Yang, Z. Zhao, C. Wang, J. Shi, D. Kim, D. Han, and D. Lo. Unveiling memorization in code models. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pages 856–856, Los Alamitos, CA, USA, apr 2024. IEEE Computer Society.

[42] Beiqi Zhang, Peng Liang, Xiyu Zhou, Aakash Ahmad, and Muhammad Waseem. Practices and challenges of using github copilot: An empirical study. In *Proceedings of the 35th International Conference on Software Engineering and Knowledge Engineering*, SEKE2023. KSI Research Inc., July 2023.

[43] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. A survey of large language models, 2023.

# Appendix A

# Extended Results

## A.1 Detailed Results for memorization for Autoregressive task

### A.1.1 Detailed Results for memorization for 32-bit

This section presents an extensive collection of results that provide deeper insights into the memorization patterns observed across different programming languages, model architectures, and experimental conditions. These detailed findings not only corroborate the main conclusions discussed in the thesis but also reveal nuanced patterns worthy of further investigation. The results for different programming languages, as shown in Figures A.1-A12, demonstrate consistent variations in memorization tendencies. Java (Figures A.1 and A.5) consistently exhibits higher levels of exact matches compared to Python (Figures A.3 and A.6), irrespective of masking ratio or model architecture. This observation aligns with our earlier discussion on the potential impact of Java's verbose syntax on memorization. The persistence of this effect across various experimental conditions underscores its significance. The impact of masking ratio on memorization is clearly illustrated in the series of graphs for each language. As the masking ratio increases from 0.1 to 0.9, we observe a general decline in exact match percentages across all languages and models. However, the rate of decline varies, with some languages and models demonstrating greater resilience to increased masking. This variance in resilience warrants further investigation and could provide insights into the robustness of different model architectures when faced with incomplete information. Interestingly, the relative performance of CodeBERT and CodeT5 remains consistent across languages, with CodeBERT generally outperforming CodeT5 in terms of exact matches. This

consistency suggests that the architectural differences between these models have a persistent effect on memorization tendencies, regardless of the programming language being processed. The distribution of memorization across different token types, as illustrated in Figures A.13-A17, reveals intriguing patterns. Variable names and string literals consistently show higher rates of exact matches compared to other token types. This finding has important implications for privacy and security considerations, as these token types are more likely to contain sensitive or identifying information. These detailed results not only support the main findings of our study but also highlight the complex interplay between model architecture, programming language characteristics, and memorization patterns in code-based LLMs. They underscore the need for nuanced, language-specific approaches to mitigating unwanted memorization and emphasize the importance of considering a wide range of experimental conditions when evaluating model performance and privacy implications.

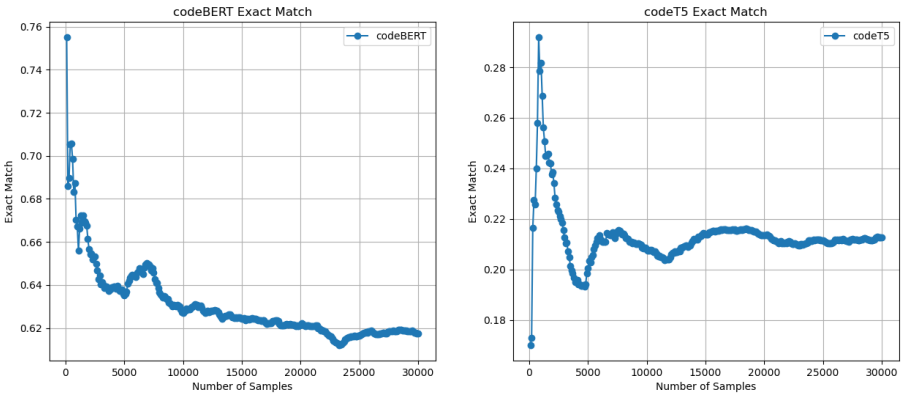


Figure A.1: memorization in 32-bit for java$_mr$0.1

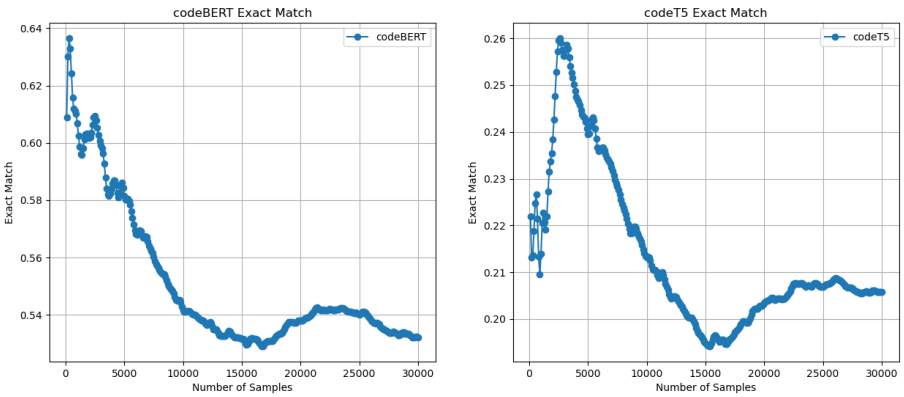


Figure A.2: memorization in 32-bit for java$_mr$0.75

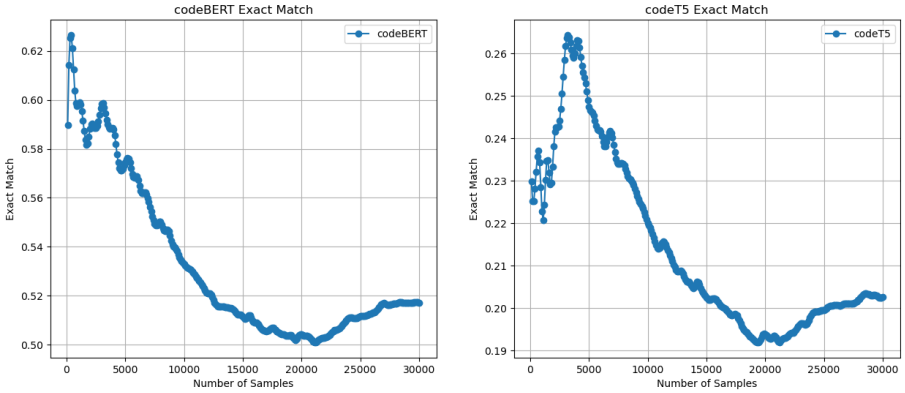


Figure A.3: memorization in 32-bit for java$_m r0.9$

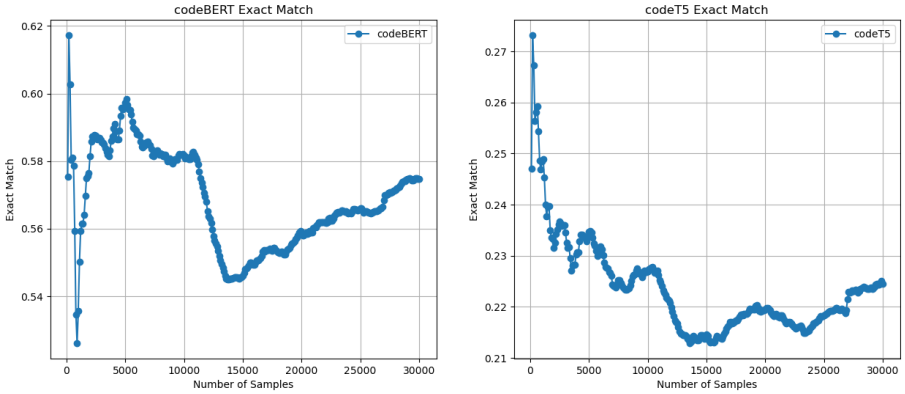


Figure A.4: memorization in 32-bit for javascript$_m r0.1$

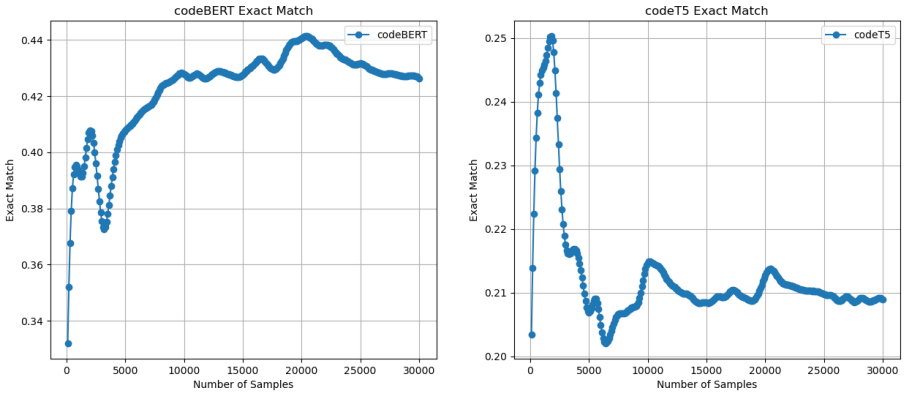


Figure A.5: memorization in 32-bit for javascript$_m r0.75$

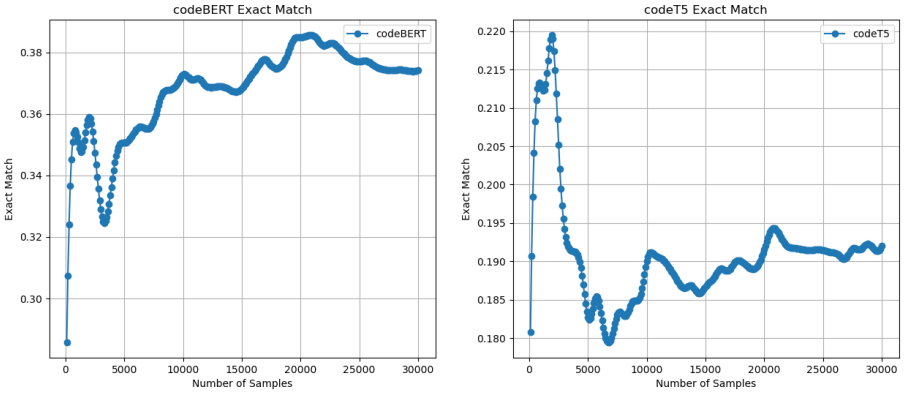


Figure A.6: memorization in 32-bit for javascript$_{mr0.9}$

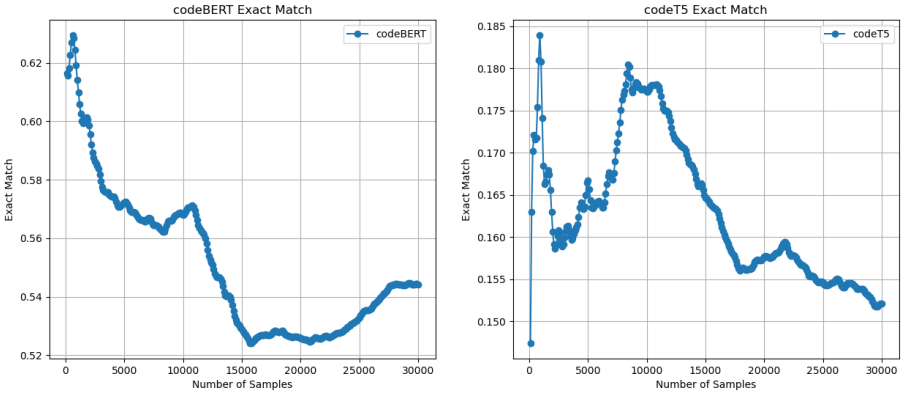


Figure A.7: memorization in 32-bit for python$_{mr0.1}$

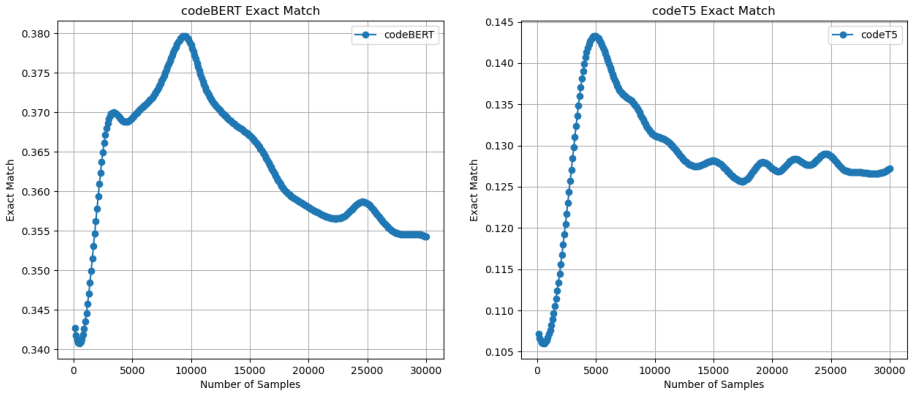


Figure A.8: memorization in 32-bit for python$_{mr0.75}$

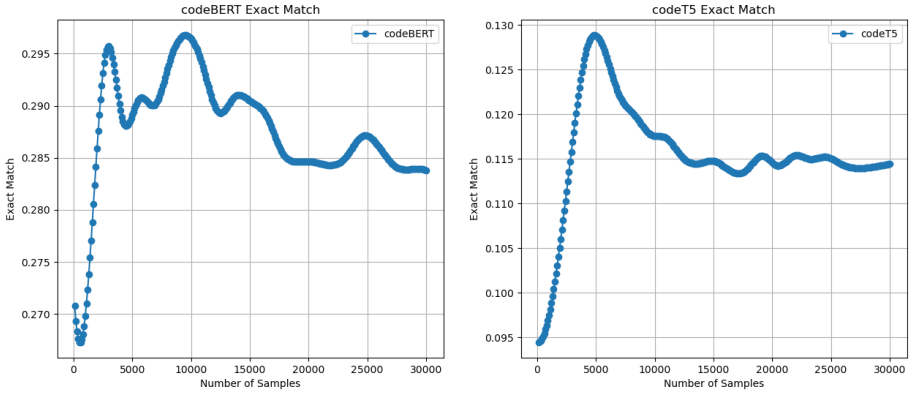


Figure A.9: memorization in 32-bit for python$_mr0.9$

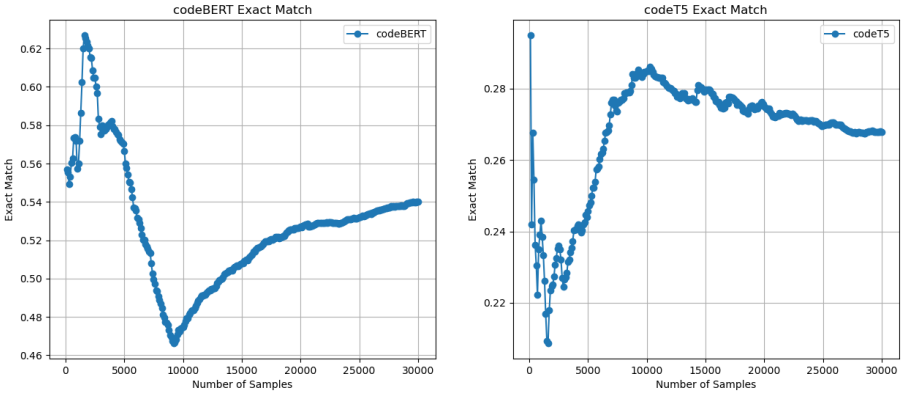


Figure A.10: memorization in 32-bit for ruby$_mr0.1$

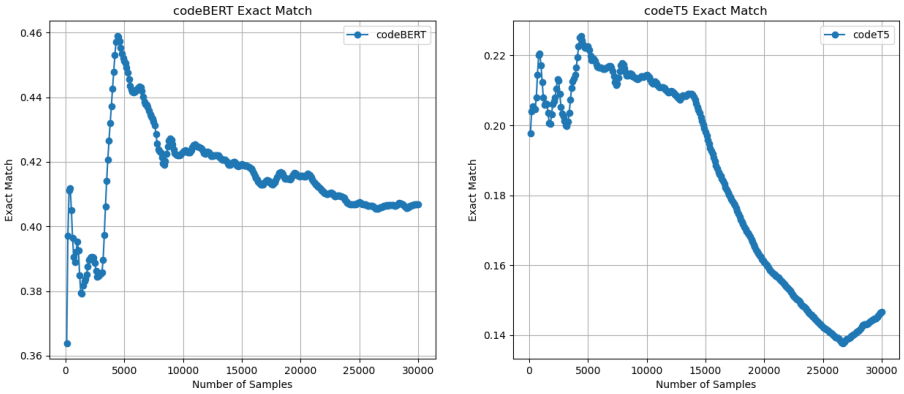


Figure A.11: memorization in 32-bit for ruby$_mr0.75$

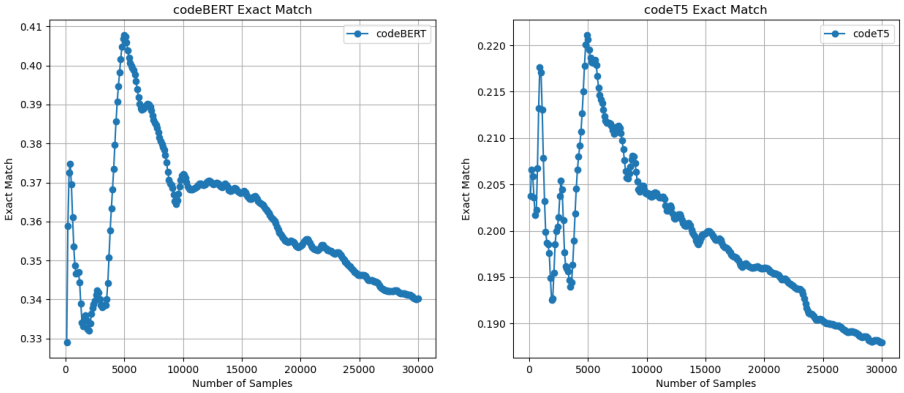


Figure A.12: memorization in 32-bit for ruby$_{mr}$0.9

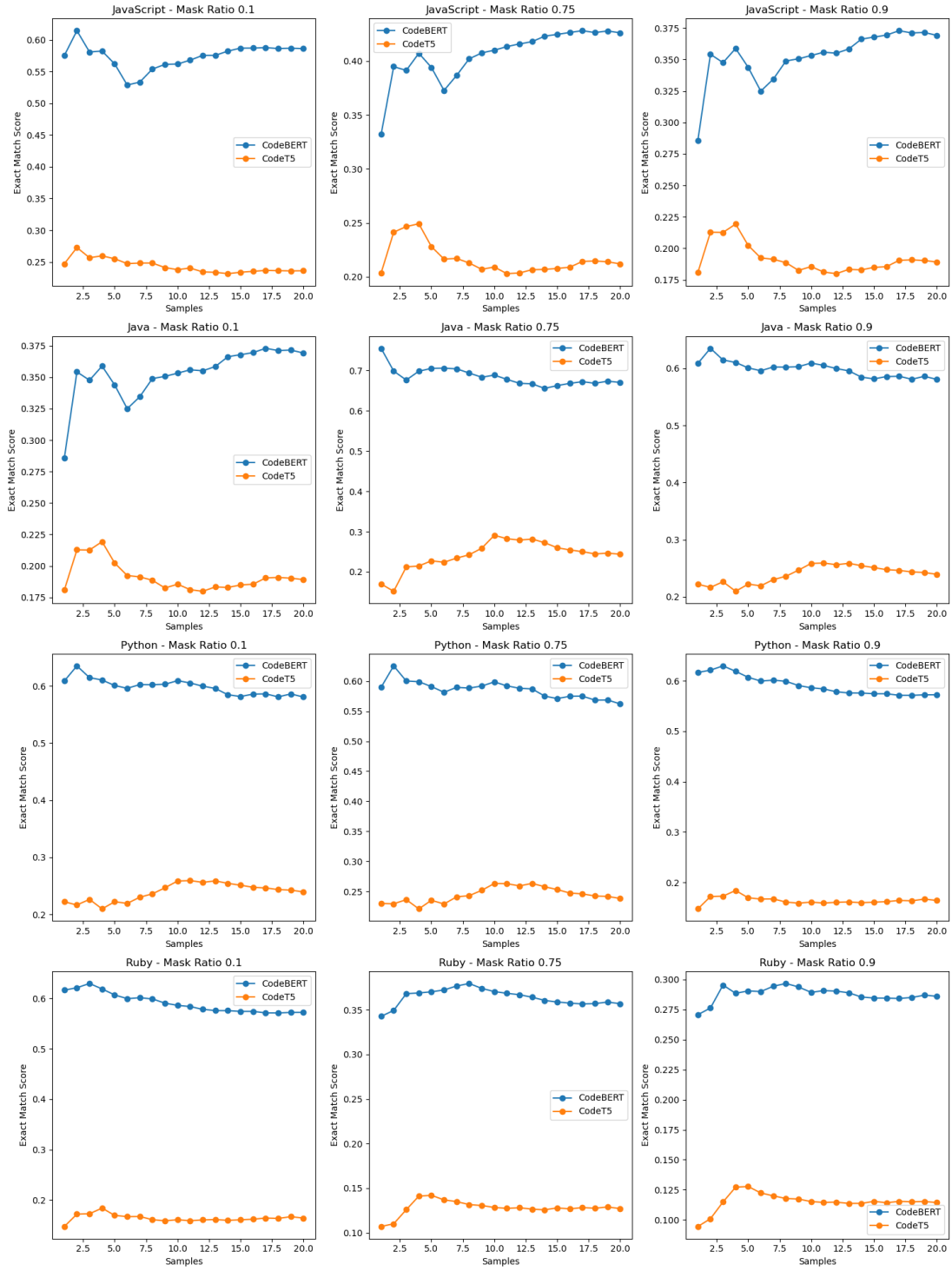


Figure A.13: Group graph for memorization in 32 bit

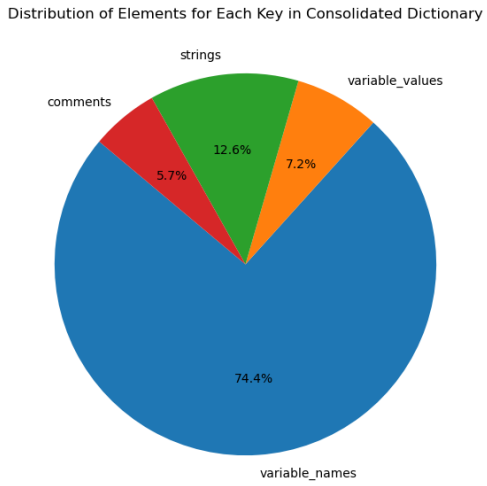


Figure A.14: Privacy Metric distribution to the memorization in Java Language

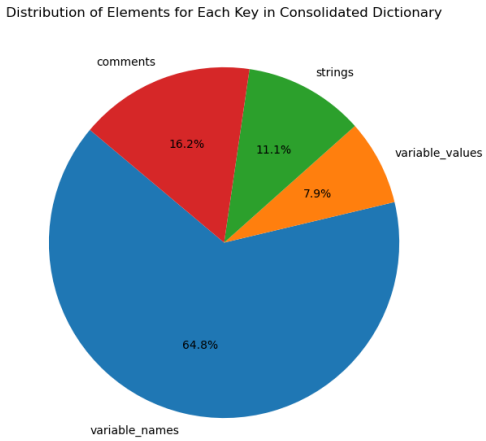


Figure A.15: Privacy Metric distribution to the memorization in JavaScript Language

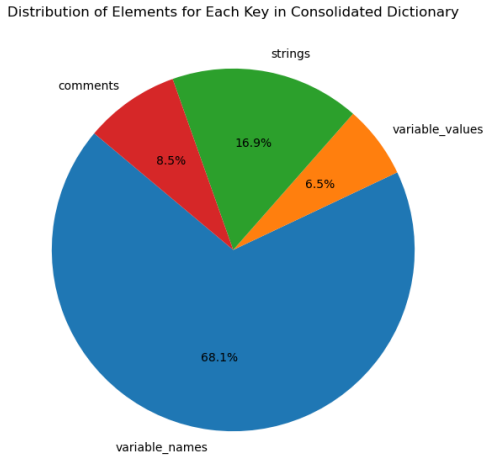


Figure A.16: Privacy Metric distribution to the memorization in Python Language

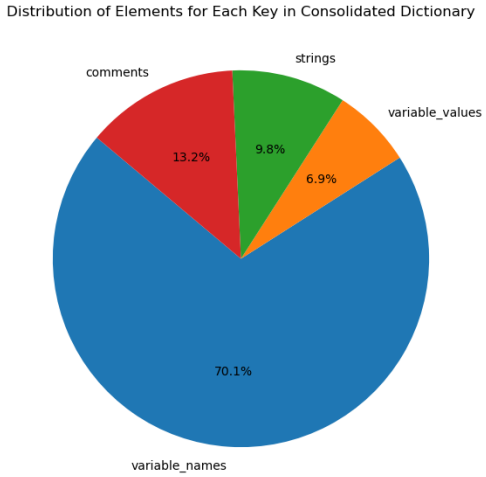


Figure A.17: Privacy Metric distribution to the memorization in Python Language

### A.1.2 Detailed Results for Memorization for 8-bit

This section presents a comprehensive analysis of memorization patterns observed in quantized models across various programming languages and experimental conditions. These results offer valuable insights into the effects of quantization on memorization tendencies in code-based LLMs.

Figures A.18-A.29 provide a detailed breakdown of memorization patterns for

quantized models across Python, Java, JavaScript, and Ruby. The impact of quantization is evident across all languages and masking ratios, with some notable trends emerging.

For Python (Figures A.18-A.20), we observe a consistent reduction in exact match percentages across all masking ratios when comparing the 8-bit quantized model to its 32-bit counterpart. This reduction is particularly pronounced at higher masking ratios, suggesting that quantization may have a more significant impact on the model's ability to memorize exact tokens when faced with heavily masked input.

Java results (Figures A.20-A.22) show a similar trend, with quantization leading to reduced memorization across all masking ratios. Interestingly, the relative difference between quantized and non-quantized models appears to be smaller for Java compared to Python, possibly indicating that the more structured nature of Java code is somewhat more resilient to the effects of quantization on memorization.

JavaScript (Figures A.23-A.25) and Ruby (Figures A.26-A.28) follow similar patterns, with quantization consistently reducing exact match percentages. However, the magnitude of this reduction varies across languages and masking ratios, highlighting the complex interplay between language characteristics, quantization, and memorization tendencies.

The group graph in Figure A.30 provides a holistic view of the impact of quantization across all languages and models. This visualization clearly demonstrates the universal trend of reduced memorization in quantized models, while also highlighting language-specific variations in the magnitude of this effect.

These detailed results offer several key insights:

1. Quantization consistently reduces exact memorization across all languages and experimental conditions, suggesting its potential as a technique for mitigating unwanted memorization in code-based LLMs.

2. The impact of quantization on memorization varies across programming languages, indicating that language-specific characteristics play a role in how quantization affects model behavior.

3. The relationship between masking ratio and the impact of quantization is non-linear, with higher masking ratios often showing a more pronounced reduction in memorization for quantized models.

4. Despite the reduction in exact memorization, quantized models still retain a significant ability to reproduce tokens from their training data, suggesting that

quantization alone may not be sufficient to completely eliminate privacy concerns related to memorization.

These findings underscore the potential of quantization as a tool for enhancing privacy in code-based LLMs while also highlighting the need for careful consideration of language-specific factors and the limitations of this approach. The results provide a foundation for further research into optimizing the trade-off between model efficiency, performance, and privacy preservation through quantization techniques.

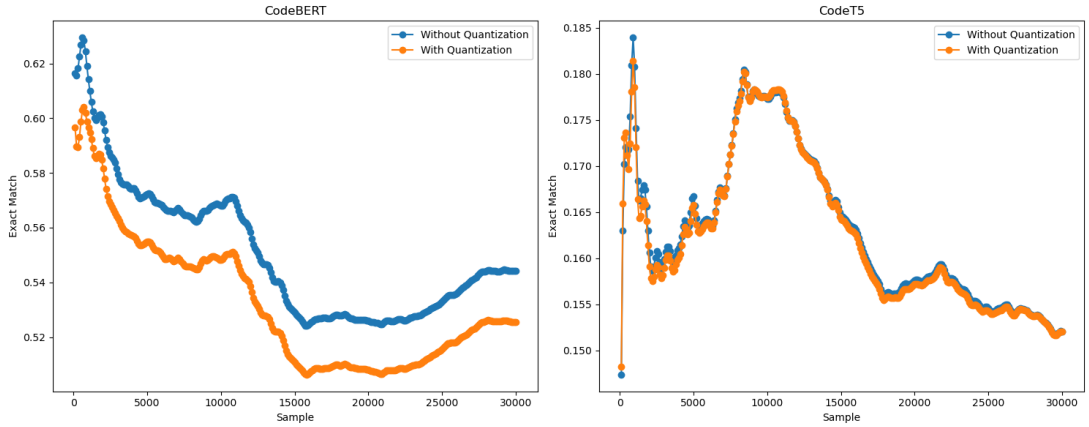


Figure A.18: memorization in 8-bit for python$_m r0.1_q False.o45080977_p ython_m r0.1_q True.o44959216$

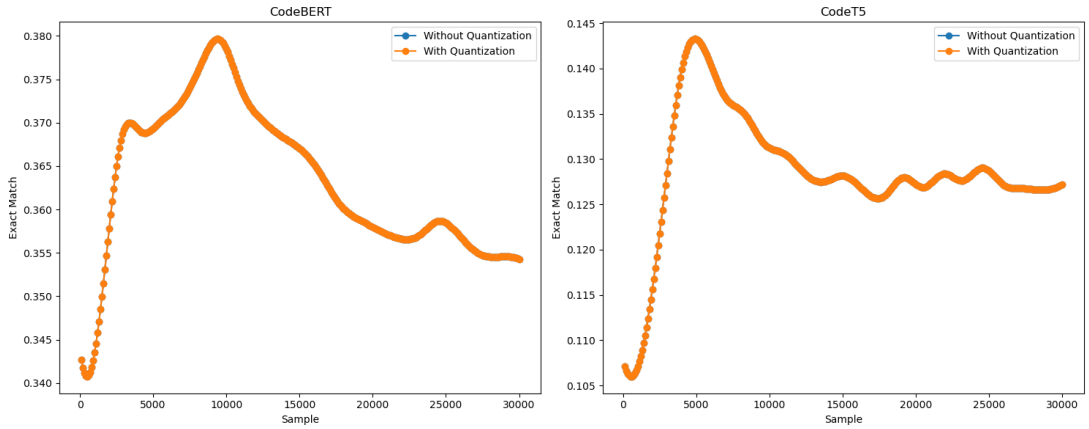


Figure A.19: memorization in 8-bit for python$_m r0.75_q False.o45080906_p ython_m r0.75_q False.o45080906$

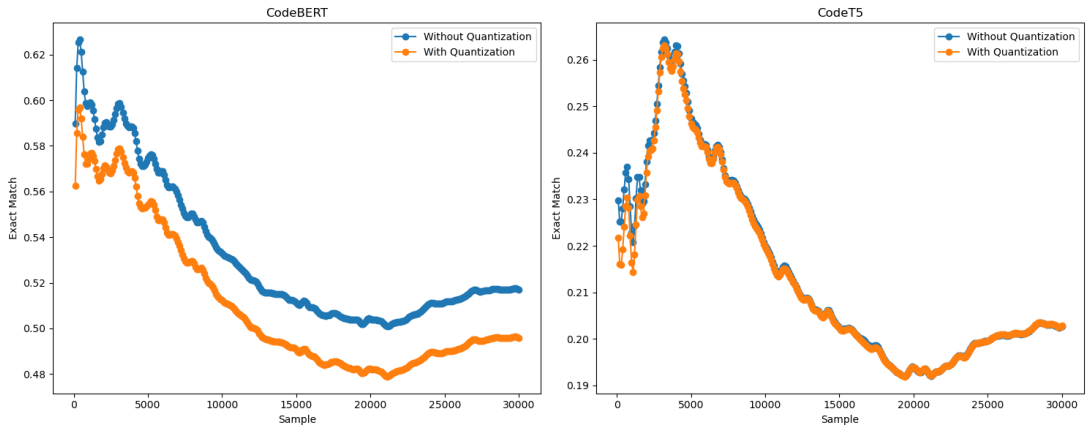


Figure A.20: memorization in 8-bit for java$_m r0.9_q False.o45080983$ $_j ava_m r0.9_q True.o44981835$

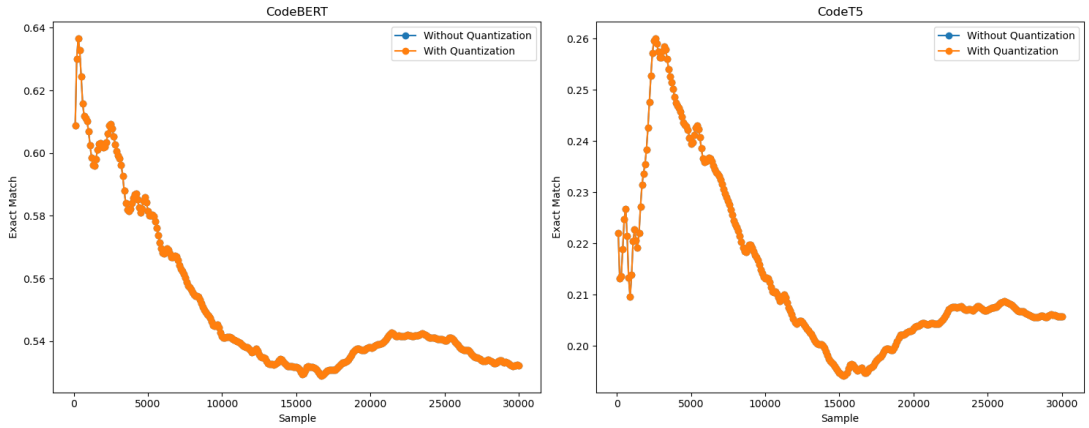


Figure A.21: memorization in 8-bit for java$_m r0.75_q False.o45080945$ $_j ava_m r0.75_q False.o45080945$

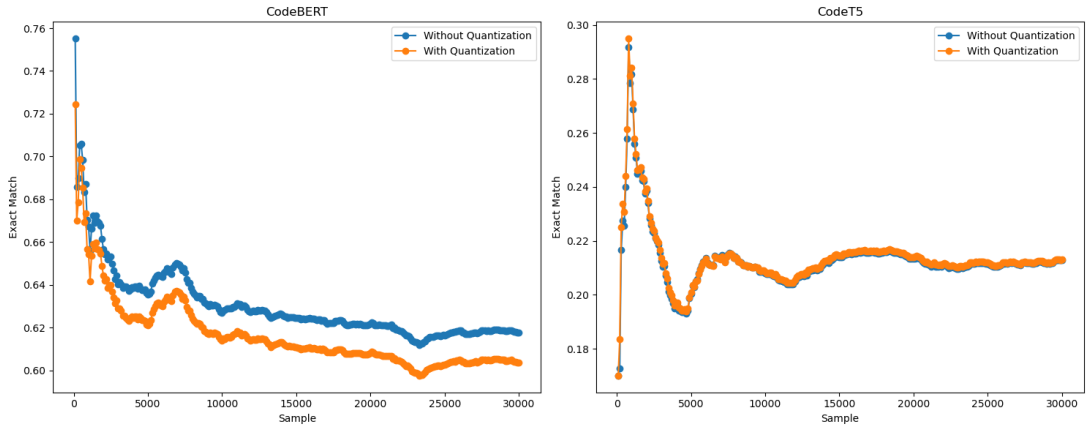


Figure A.22: memorization in 8-bit for java$_m r0.1_q False.o45080974$ $_j ava_m r0.1_q True.o44981833$

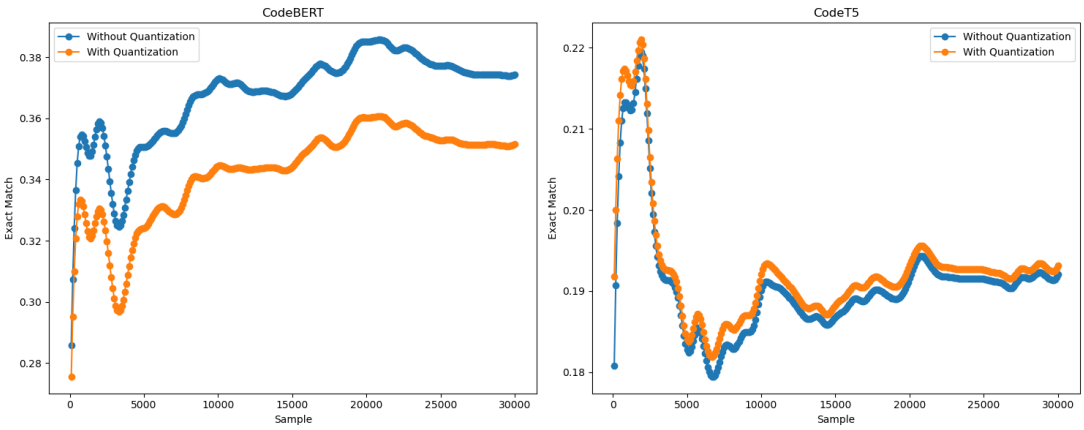


Figure A.23: memorization in 8-bit for javascript$_m r0.9_q False.o45080999\,javascript_m r0.9_q True.o449818$

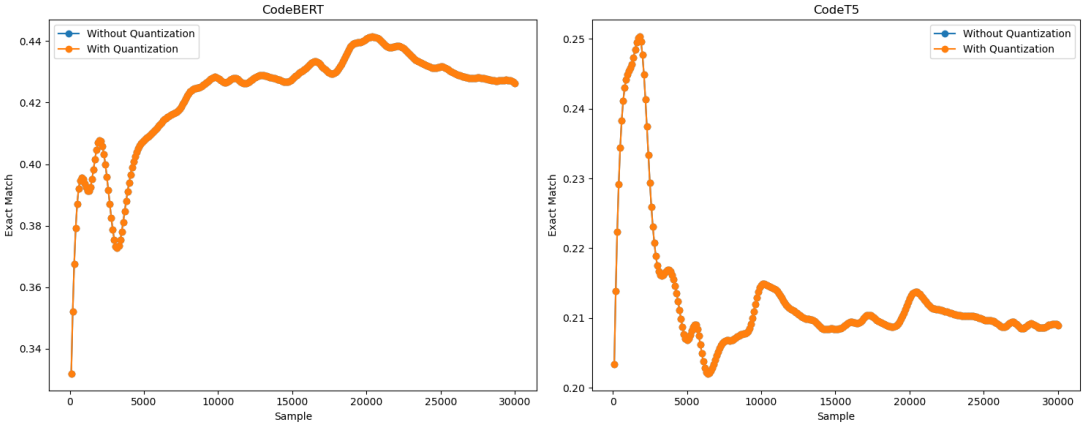


Figure A.24: memorization in 8-bit for javascript$_m r0.75_q False.o45080948\,javascript_m r0.75_q False.o450$

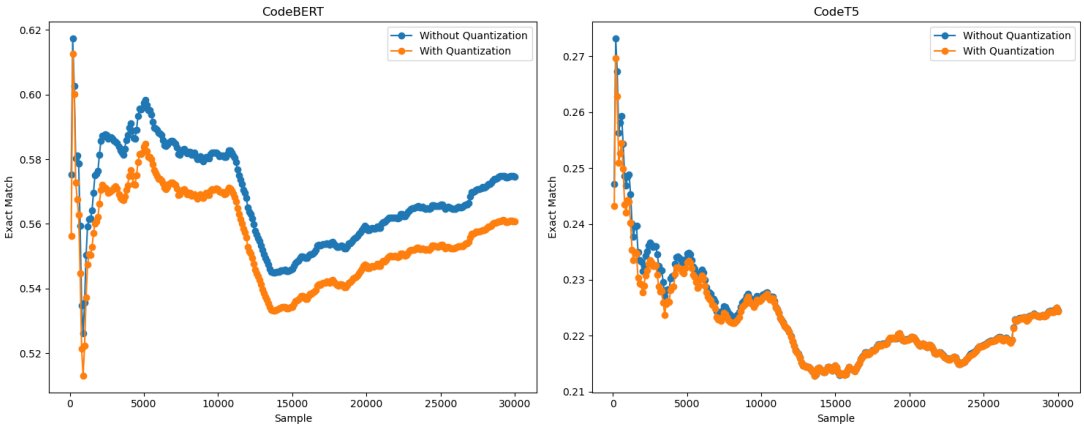


Figure A.25: memorization in 8-bit for javascript$_m r0.1_q False.o45080970\,javascript_m r0.1_q True.o449592$

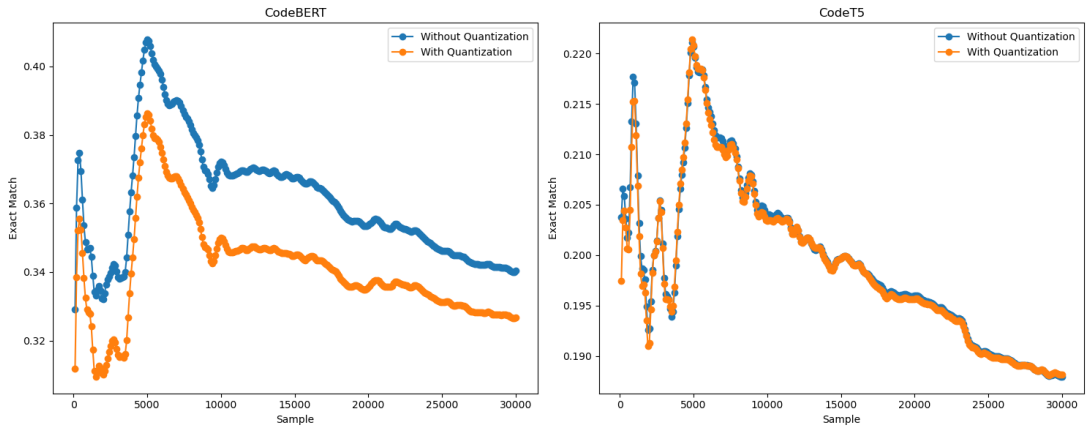


Figure A.26: memorization in 8-bit for ruby$_m r0.9_q False.o45081007_r uby_m r0.9_q True.o44981858$

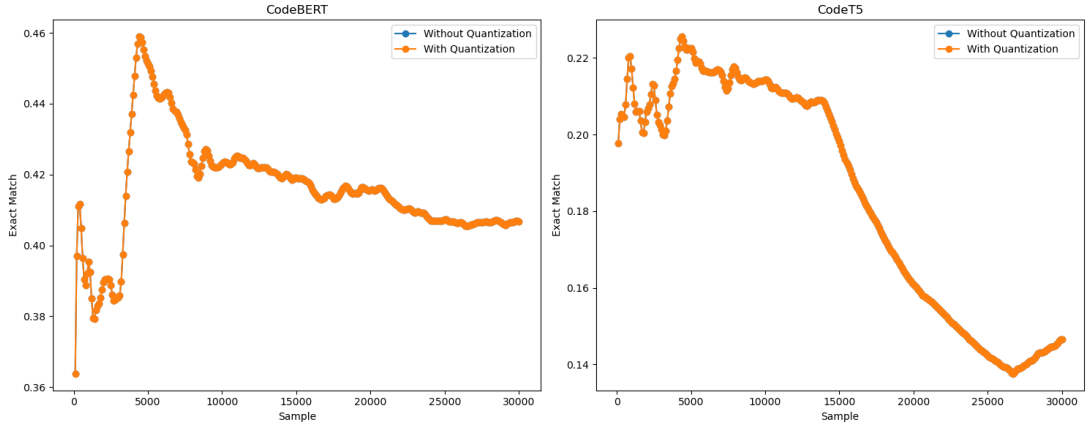


Figure A.27: memorization in 8-bit for ruby$_m r0.75_q False.o45080954_r uby_m r0.75_q False.o45080954$

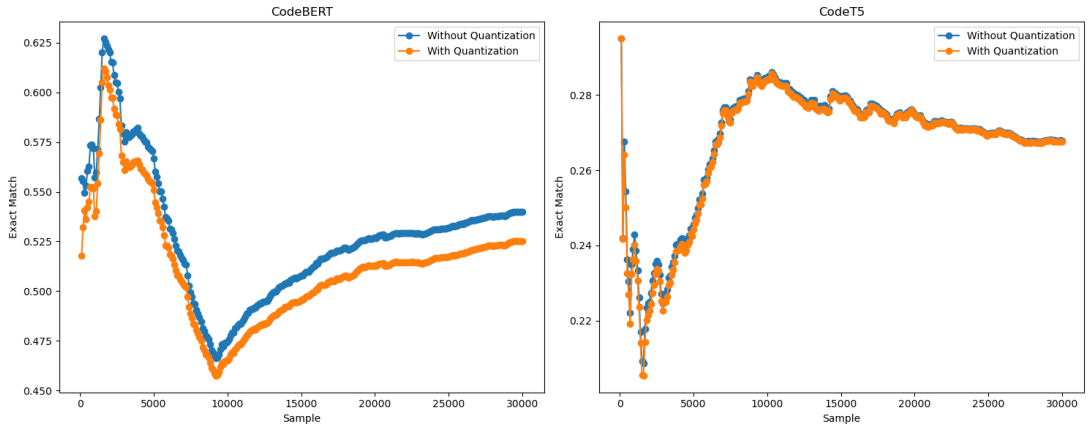


Figure A.28: memorization in 8-bit for ruby$_m r0.1_q False.o45080965_r uby_m r0.1_q True.o44981857$

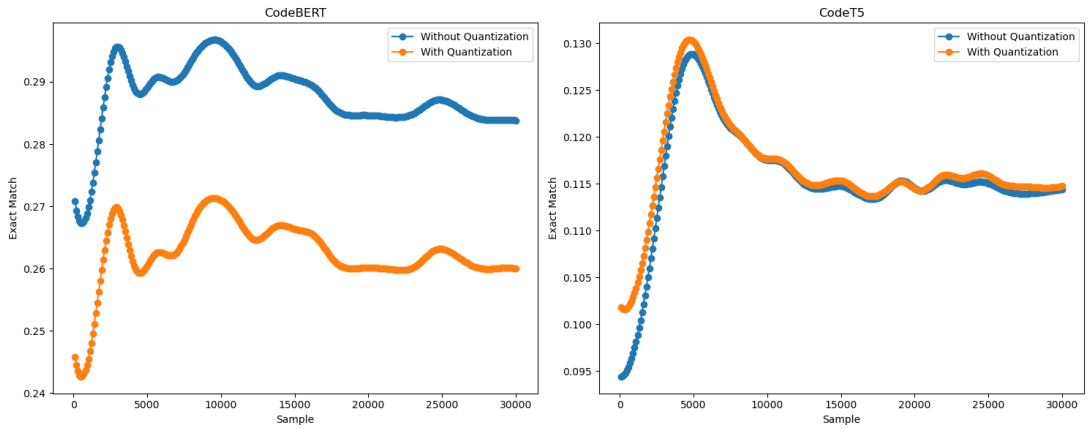


Figure A.29: memorization in 8-bit for $\text{python}_{m}r0.9_{q}False.o45080980_{p}ython_{m}r0.9_{q}True.o44959232$

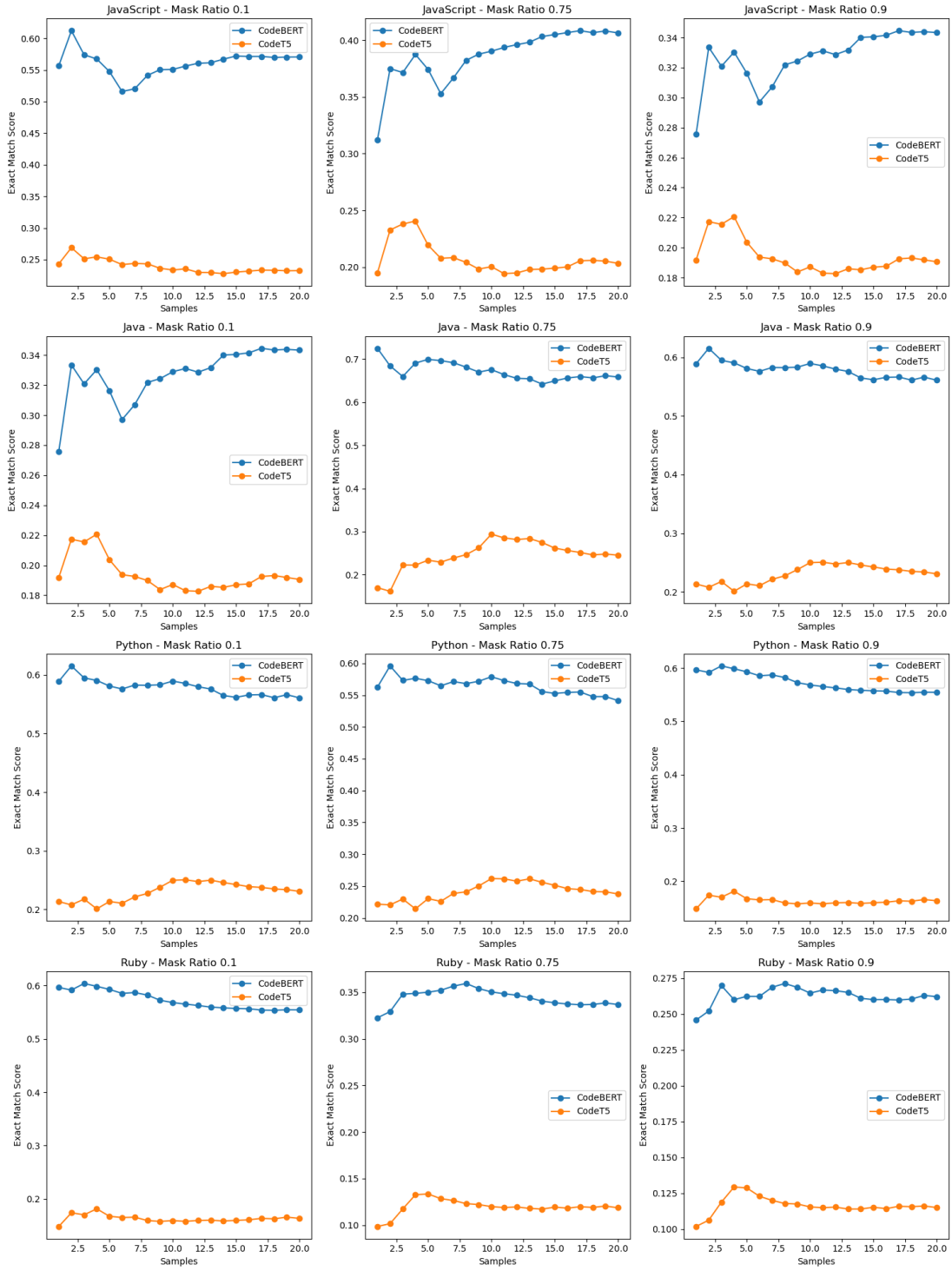


Figure A.30: Group Graph quantized

## A.2 Detailed Results for memorization for Autoregressive task

This section provides an in-depth analysis of memorization patterns observed in autoregressive models across different programming languages and quantization levels. These results offer valuable insights into how autoregressive models, specifically CodeGPT and CodeT5, memorize and generate code in various contexts.

Figures A.31-A.38 present a comprehensive breakdown of memorization patterns for both 8-bit quantized and 32-bit models across Ruby, JavaScript, Python, and Java. The results reveal intriguing patterns and differences between languages and quantization levels.

For Ruby (Figures A.31 and A.35), we observe that both CodeGPT and CodeT5 exhibit similar memorization tendencies, with CodeGPT showing slightly higher CodeBLEU scores across most samples. The impact of quantization appears to be minimal for Ruby, with the 8-bit and 32-bit models performing comparably.

JavaScript results (Figures A.32 and A.36) demonstrate a more pronounced difference between CodeGPT and CodeT5, with CodeGPT consistently achieving higher CodeBLEU scores. This suggests that CodeGPT may be more adept at memorizing and reproducing JavaScript code patterns. The effect of quantization on JavaScript is more noticeable than in Ruby, with a slight reduction in CodeBLEU scores for the 8-bit models.

Python (Figures A.33 and A.37) shows an interesting pattern where the difference between CodeGPT and CodeT5 performance is less pronounced compared to other languages. This could indicate that both models are well-optimized for Python code generation. The impact of quantization on Python appears to be moderate, with a small but consistent reduction in CodeBLEU scores for 8-bit models.

Java results (Figures A.34 and A.38) reveal the highest overall CodeBLEU scores among all languages, suggesting that both CodeGPT and CodeT5 are particularly effective at memorizing and generating Java code. The gap between CodeGPT and CodeT5 performance is also more pronounced for Java. Interestingly, the impact of quantization on Java memorization appears to be the least significant among all languages examined.

These detailed results provide several key insights:

1. Language-specific variations: The extent of memorization and the relative per-

formance of CodeGPT versus CodeT5 vary significantly across programming languages. This underscores the importance of language-specific considerations in model development and evaluation.

2. Model architecture impact: CodeGPT consistently outperforms CodeT5 in terms of CodeBLEU scores across all languages, suggesting that its architecture may be more conducive to memorizing and reproducing code patterns in autoregressive tasks.

3. Quantization effects: The impact of 8-bit quantization on memorization in autoregressive tasks appears to be less pronounced compared to the masked language modeling tasks discussed earlier. This suggests that autoregressive models may be more resilient to quantization in terms of their memorization capabilities.

4. Java optimization: Both models demonstrate particularly high memorization tendencies for Java, indicating that they may be especially well-optimized for this language or that Java's structure lends itself well to memorization by these models.

5. Consistency across samples: The relative performance of models and the impact of quantization remain fairly consistent across different code samples within each language, suggesting that these patterns are robust and not heavily influenced by individual sample characteristics.

These findings provide valuable insights into the memorization behaviors of autoregressive code models and how they vary across languages and quantization levels. They highlight the complex interplay between model architecture, programming language characteristics, and quantization in determining memorization tendencies. These results can inform future research into optimizing code generation models for different languages and use cases, as well as guide efforts to balance performance, efficiency, and privacy considerations in autoregressive code models.

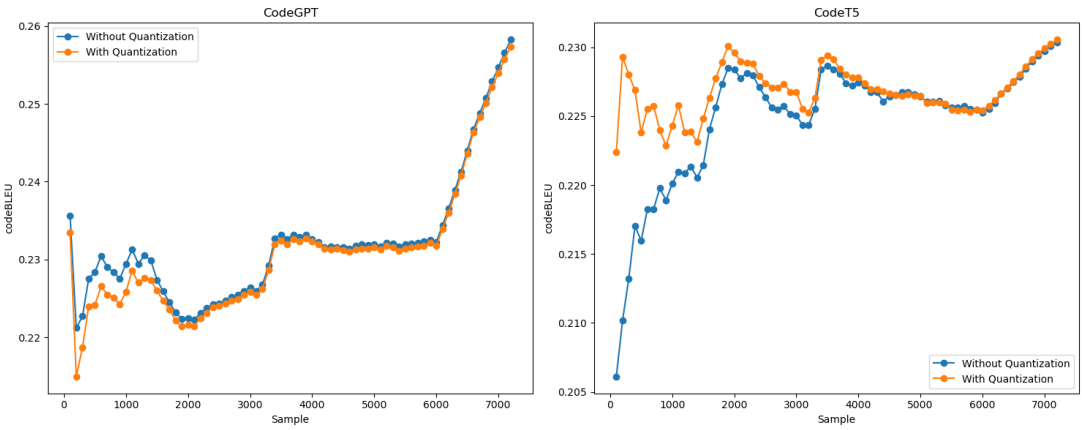


Figure A.31: memorization for NTP task for ruby8

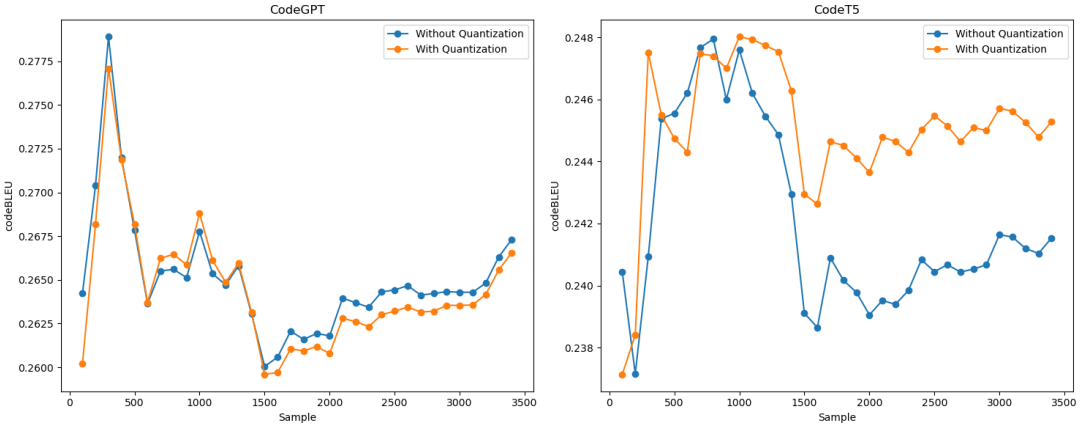


Figure A.32: memorization for NTP task for javascript8

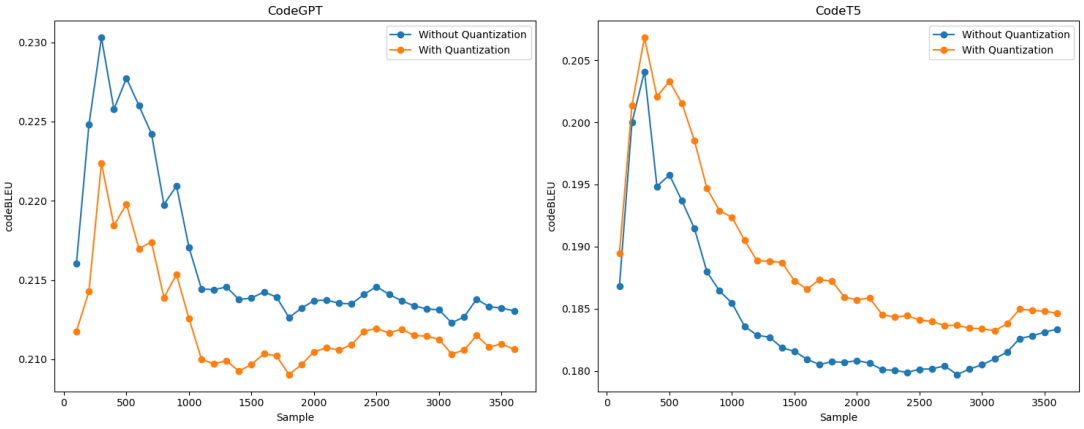


Figure A.33: memorization for NTP task for python8

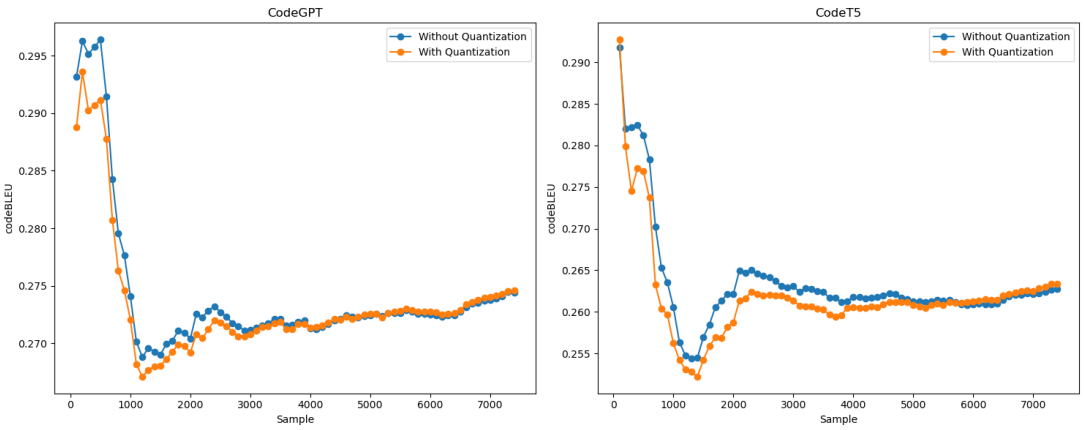


Figure A.34: memorization for NTP task for java8

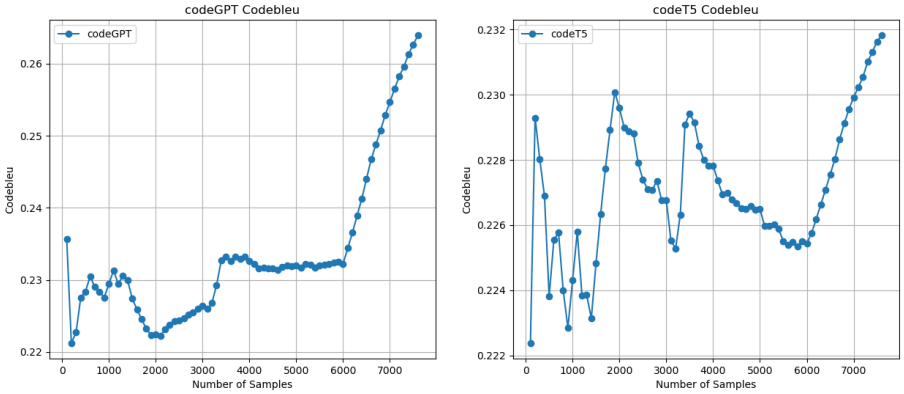


Figure A.35: memorization for NTP task for ruby32

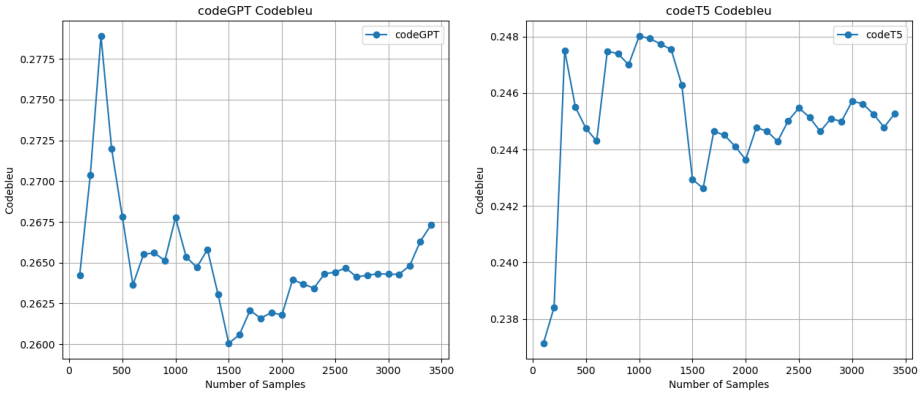


Figure A.36: memorization for NTP task for javascript32

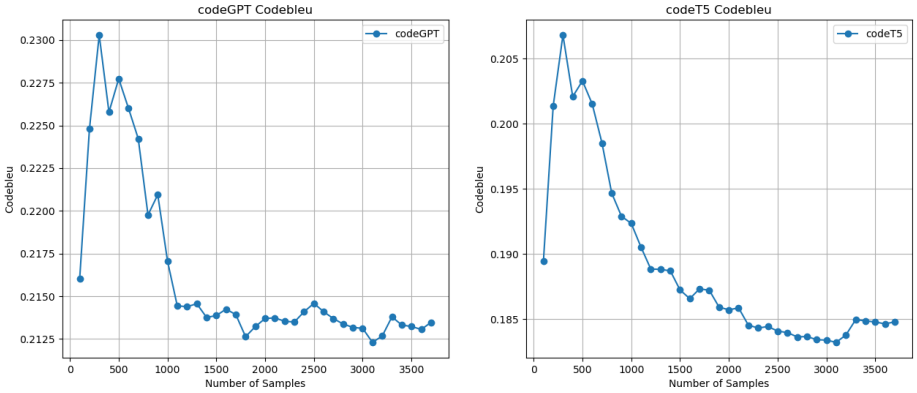


Figure A.37: memorization for NTP task for python32

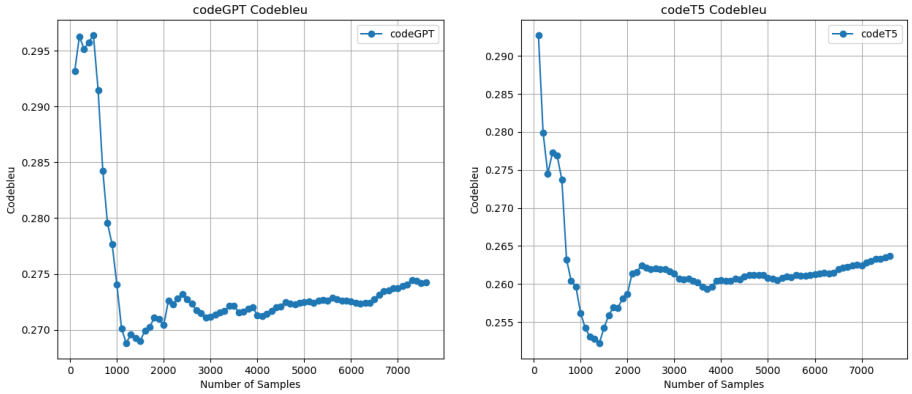


Figure A.38: memorization for NTP task for java32

## A.3 Few-Shot for Autoregressive Task

This section presents an in-depth analysis of how Few-Shot Learning affects memorization patterns in autoregressive code generation models. This investigation provides crucial insights into the adaptability of these models and how they balance memorization with generalization when presented with a small number of examples.

Figures A.39 and A.40 illustrate the performance of CodeGPT and CodeT5 respectively under Few-Shot Learning conditions for Java code generation. These results reveal several intriguing patterns and implications for Few-Shot Learning in code generation tasks.

The graphs show considerable variation in CodeBLEU scores across different numbers of shots (examples provided) and across different code samples. This variability

suggests that the effectiveness of Few-Shot Learning in code generation is highly context-dependent and can be influenced by factors such as the complexity of the task, the relevance of the provided examples, and the specific characteristics of the code being generated.

For CodeGPT (Figure A.39), we observe that increasing the number of shots does not always lead to improved performance. In some cases, providing more examples actually results in lower CodeBLEU scores. This non-monotonic relationship between the number of shots and performance suggests that the model's ability to leverage few-shot examples is nuanced and not simply a matter of "more is better."

CodeT5 (Figure A.40) shows a similar pattern of variability, but with some notable differences from CodeGPT. The range of CodeBLEU scores for CodeT5 appears to be somewhat narrower, suggesting that it might be more consistent in its Few-Shot Learning performance across different samples. However, it too shows instances where additional shots lead to decreased performance.

These observations highlight several key points:

1. Context Sensitivity: The effectiveness of Few-Shot Learning in code generation tasks is highly dependent on the specific context and examples provided. This underscores the importance of careful example selection in few-shot scenarios.

2. Overfitting Risk: The decrease in performance with additional shots in some cases suggests a risk of overfitting to the provided examples. This indicates that models may sometimes prioritize mimicking the few-shot examples over generalizing to the broader task.

3. Model Differences: The different patterns observed between CodeGPT and CodeT5 hint at architectural differences in how these models process and utilize few-shot examples. This suggests that different model architectures may require different approaches to optimize Few-Shot Learning in code generation tasks.

4. Memorization vs. Generalization: The variable performance across different numbers of shots reflects the complex balance between memorization of provided examples and generalization to new contexts. This balance appears to shift dynamically based on the specific examples and task at hand.

5. Practical Implications: The inconsistent relationship between the number of shots and performance has important implications for the practical application

of Few-Shot Learning in code generation. It suggests that simply providing more examples may not always be the best strategy, and that careful curation of examples may be necessary for optimal performance.

To illustrate these points further, the section includes specific examples (Example 1 to Example 4) that demonstrate how different prompts can lead to significantly different CodeBLEU scores. These examples provide concrete instances of how the choice and presentation of few-shot examples can dramatically impact the model's output.

In conclusion, this analysis of Few-Shot Learning in autoregressive code generation tasks reveals a complex and nuanced picture. While Few-Shot Learning can indeed enhance the performance of these models, its effectiveness is highly variable and context-dependent. These findings emphasize the need for sophisticated strategies in applying Few-Shot Learning to code generation tasks, taking into account the specific characteristics of the task, the model architecture, and the nature of the provided examples. Future research in this area could focus on developing more robust methods for example selection and prompt engineering to optimize few-shot performance in code generation models.

### Sample examples for oscillating performance: Example sensitivity

```
Example 1:
Prefix: public List<ExtendedRelation> getRelations(String baseTable,
                  String baseColumn, String relatedTable, String
                      relatedColumn,
                  String relation, String mappingTable) throws
                      SQLException {

            List<ExtendedRelation> relations = null;

            try {
                  if (extendedRelationsDao.isTableExists()) {
                        relations = extendedRelationsDao.getRelations
                            (baseTable,
                                    baseColumn, relatedTable,
                                        relatedColumn, relation,
                                    mappingTable);
Suffix: } else {
```

```
                        relations = new ArrayList<>();
                    }
                } catch (SQLException e) {
                    throw new GeoPackageException(
                            "Failed to get relationships. Base
                                Table: " + baseTable
                                    + ", Base Column: " +
                                        baseColumn
                                    + ", Related Table: " +
                                        relatedTable
                                    + ", Related Column: " +
                                         relatedColumn
                                    + ", Relation: " +
                                        relation + ",
                                        Mapping Table: "
                                    + mappingTable, e);
                }

                return relations;
            }
Prefix: public void inquire(String vaultName, String resourceGroupName,
    String fabricName, String containerName) {
Suffix:
```

This prompt passed onto codeGPT gave an codeBLEU score of 37.29%.

whereas,

```
    Example 2:
Prefix: public String getStringUtf8(final int index)
    {
        boundsCheck0(index, SIZE_OF_INT);

        final int
Suffix: length = UNSAFE.getInt(byteArray, ARRAY_BASE_OFFSET + index);

        return getStringUtf8(index, length);
    }
Prefix: public void inquire(String vaultName, String resourceGroupName,
    String fabricName, String containerName) {
```

```
Suffix:
```

This prompt passed onto codeGPT gave a codeBLEU score of 26.80%.

### Sample examples to show oscillating performance - order sensitivity

```
    Example 3
    Prefix: @Override
    public boolean exist(IESigType esigType, String id) {
Suffix: return eSigList.indexOf(esigType, id) >= 0;
    }
Prefix: public static Map<String, String> copyMap(Map<String, String>
    originalMap) {
        Objects.requireNonNull(originalMap, Required.MAP.toString());
Suffix:
        return new HashMap<>(originalMap);
    }
Prefix: public void setLong(Long value) {
                this.checkNull(value);
Suffix: list.add(value);
                type.add(Long.class);
        }
Prefix: @Override
    public IPermissionTarget getTarget(String key) {

        /*
         * If the specified key matches one of the "all entity" style
             targets,
         * just return the appropriate target.
         */
        switch (key) {
            case IPermission.ALL_CATEGORIES_TARGET:
                return ALL_CATEGORIES_TARGET;
            case IPermission.ALL_PORTLETS_TARGET:
                return ALL_PORTLETS_TARGET;
            case IPermission.ALL_GROUPS_TARGET:
                return ALL_GROUPS_TARGET;
                // Else just fall through...
```

```
        }

Suffix:
```

This prompt passed in CodeGPT gave a codeBLEU of 26.44%

```
    Example 4
    Prefix: public void setLong(Long value) {
                this.checkNull(value);
Suffix: list.add(value);
                type.add(Long.class);
        }
Prefix: public static Map<String, String> copyMap(Map<String, String>
    originalMap) {
        Objects.requireNonNull(originalMap, Required.MAP.toString());
Suffix:
        return new HashMap<>(originalMap);
    }
Prefix: @Override
    public boolean exist(IESigType esigType, String id) {
Suffix: return eSigList.indexOf(esigType, id) >= 0;
    }
Prefix: @Override
    public IPermissionTarget getTarget(String key) {

        /*
         * If the specified key matches one of the "all entity" style
            targets,
         * just return the appropriate target.
         */
        switch (key) {
            case IPermission.ALL_CATEGORIES_TARGET:
                return ALL_CATEGORIES_TARGET;
            case IPermission.ALL_PORTLETS_TARGET:
                return ALL_PORTLETS_TARGET;
            case IPermission.ALL_GROUPS_TARGET:
                return ALL_GROUPS_TARGET;
                // Else just fall through...
        }
```

`Suffix:`

This prompt passed onto codeGPT gave a codeBLEU score of 21.65%

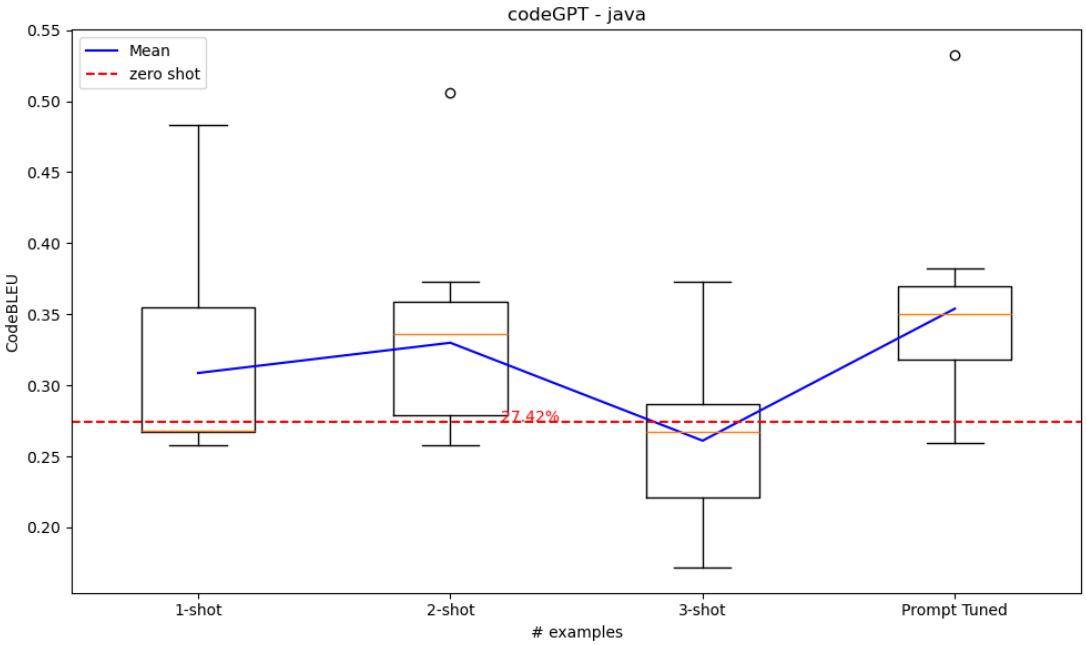


Figure A.39: fslautogptjava

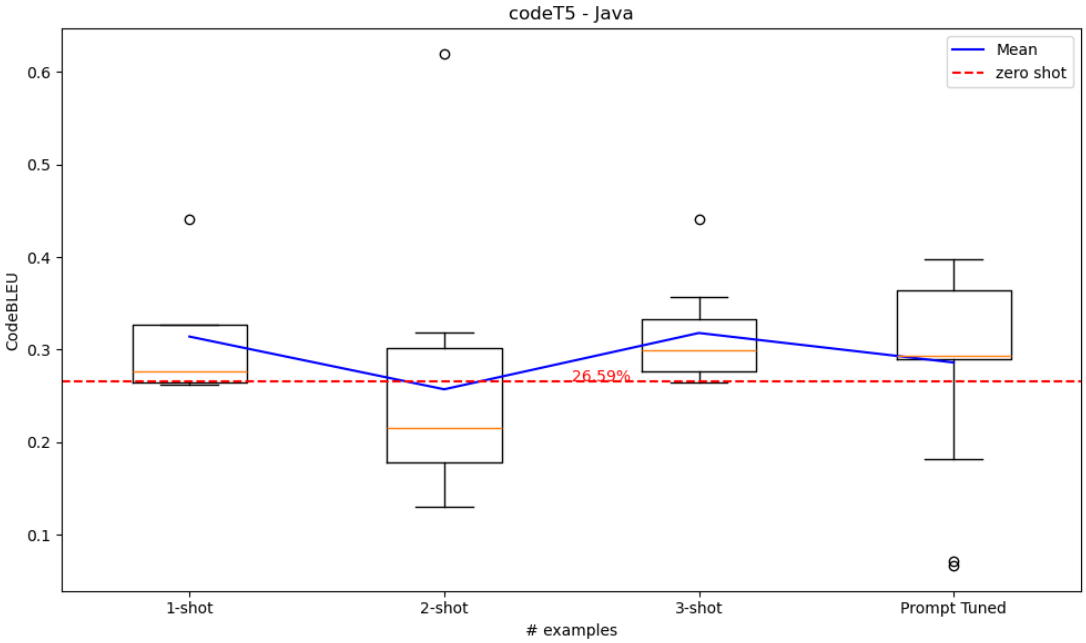


Figure A.40: fslautot5java

## A.4 Few-Shot for Non-Autoregressive Task

This section explores the impact of Few-Shot Learning on memorization patterns in non-autoregressive code models, specifically focusing on masked language modeling (MLM) tasks. This analysis provides valuable insights into how these models adapt to and utilize a small number of examples in a different paradigm from autoregressive generation.

Figures A.41 and A.42 present the results of Few-Shot Learning experiments for CodeT5 and CodeBERT respectively, applied to MLM tasks. These visualizations reveal intriguing patterns that both align with and diverge from what we observed in the autoregressive tasks.

For CodeT5 (Figure A.41), we see a complex relationship between the number of shots and the exact match percentage. The graph shows considerable variability across different code samples and shot numbers. Notably, there are instances where increasing the number of shots leads to improved performance, but also cases where it results in decreased accuracy. This non-linear relationship suggests that the model's ability to leverage few-shot examples in MLM tasks is highly context-dependent.

CodeBERT (Figure A.42) exhibits a somewhat different pattern. While there is still variability across samples, the overall trend appears to be more consistent, with a general increase in exact match percentage as the number of shots increases. This suggests that CodeBERT may be more adept at leveraging few-shot examples in MLM tasks compared to CodeT5.

These observations highlight several key points:

1. Task-Specific Differences: The patterns observed in MLM tasks differ from those seen in autoregressive tasks, underscoring the importance of considering task-specific characteristics when implementing Few-Shot Learning in code models.

2. Model Architecture Impact: The different patterns exhibited by CodeT5 and CodeBERT suggest that model architecture plays a significant role in how effectively Few-Shot Learning can be applied to MLM tasks.

3. Consistency vs. Variability: CodeBERT's more consistent improvement with additional shots contrasts with CodeT5's higher variability, indicating that different models may require different strategies for optimal Few-Shot Learning in MLM tasks.

4. Context Sensitivity: The variability across different code samples reinforces the notion that the effectiveness of Few-Shot Learning is highly dependent on the specific context and nature of the code being processed.

5. Potential for Overfitting: While CodeBERT shows a general trend of improvement with more shots, the occasional decreases in performance for both models suggest that there's still a risk of overfitting to the provided examples.

6. Implications for Memorization: The varying effectiveness of Few-Shot Learning in MLM tasks has important implications for understanding how these models memorize and generalize code patterns. It suggests that memorization in MLM tasks may be more directly influenced by few-shot examples compared to autoregressive tasks.

These findings have several important implications for the application of Few-Shot Learning in non-autoregressive code models:

1. They highlight the need for careful consideration of model architecture when designing Few-Shot Learning strategies for MLM tasks in code processing.

2. The results suggest that optimizing Few-Shot Learning for MLM tasks may require different approaches compared to autoregressive tasks, potentially involving more focused example selection or task-specific prompt engineering.

3. The variability in performance across different samples emphasizes the importance of robust evaluation methods that consider a wide range of code contexts when assessing Few-Shot Learning effectiveness in MLM tasks.

The Prompts used for Few-Shot Learning is as follows:

```
Prefix: masked prefix code
Suffix: masked values
Prefix: masked prefix code
Suffix:
```

In conclusion, this analysis of Few-Shot Learning in non-autoregressive code tasks reveals a complex interplay between model architecture, task characteristics, and the nature of the provided examples. While Few-Shot Learning shows promise for improving performance in MLM tasks, its effectiveness varies significantly based on these factors.

These findings open up new avenues for research into optimizing Few-Shot Learning strategies for different types of code processing tasks and model architectures. Future work could focus on developing more sophisticated methods for example selection and prompt design specifically tailored to MLM tasks in code models, as well as investigating the underlying mechanisms that lead to the observed differences between models like CodeT5 and CodeBERT in their Few-Shot Learning capabilities.

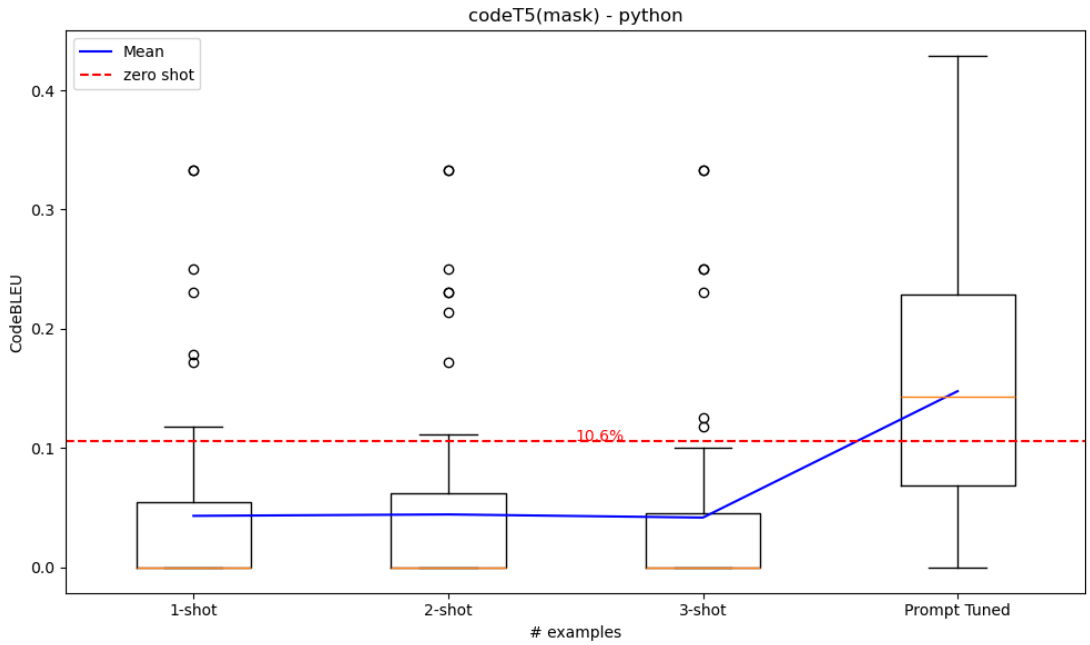


Figure A.41: fsl-mlm-t5

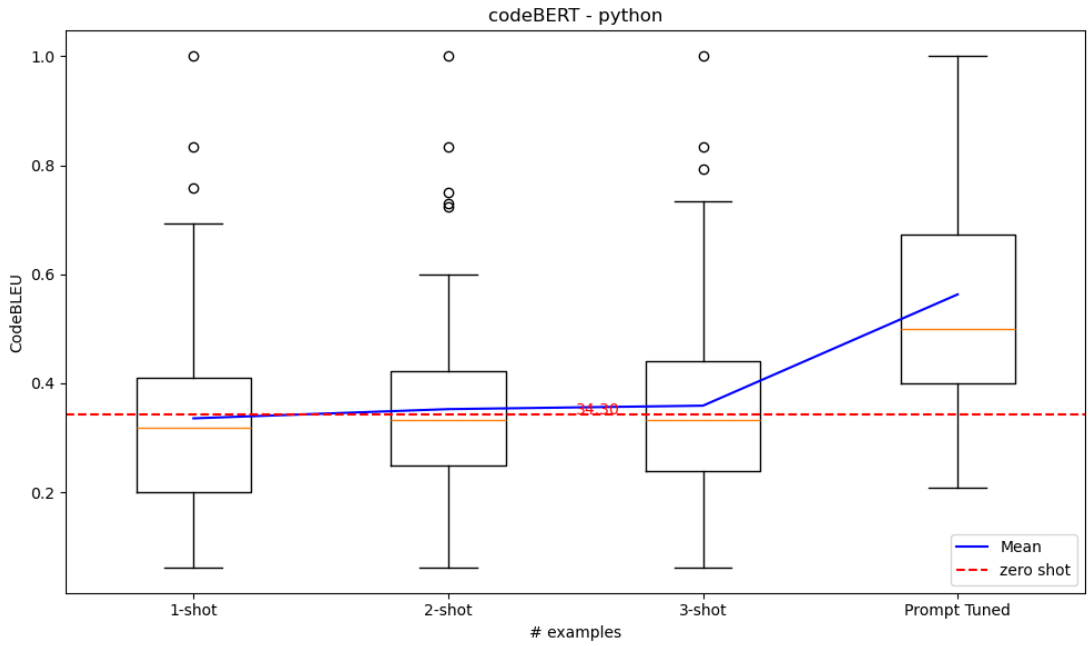


Figure A.42: fslmlmbert

## A.5 Dataset

The dataset used in this study exhibits significant diversity in terms of code length, functionality, and complexity across different programming languages. Figure A.43 showcases dataset samples from different indices, highlighting the varying nature of code with respect to length and functionality. This variation in the dataset contributes to the proportional variation in memorization patterns observed in our results, as illustrated in Figure 4.5, which corresponds specifically to Java code samples. Further analysis of the dataset distribution across multiple programming languages reveals notable differences in code length characteristics. Figures A.44, A.45, A.46, and A.47 present the dataset distributions for Python, Java, JavaScript, and Ruby, respectively. These distributions indicate substantial variations in mean program length across languages. For instance, Python code samples have a mean length of 1056 characters, whereas Ruby samples have a much shorter mean length of just 461 characters. This significant disparity in average code length between languages like Python and Ruby provides insight into the memorization patterns discussed in Chapter 4. Despite Ruby's smaller representation in the dataset, the relatively shorter length of its code samples may contribute to the smaller difference in memorization rates compared to more prevalent languages like Python. This observation underscores the complex relationship between

code characteristics, dataset composition, and memorization tendencies in large language models for code. The diversity in code length and complexity across languages emphasizes the importance of considering language-specific features when analyzing and interpreting memorization patterns in code-based language models. It also highlights the need for careful dataset curation and analysis to ensure fair comparisons across different programming languages and to accurately assess model performance and memorization tendencies.

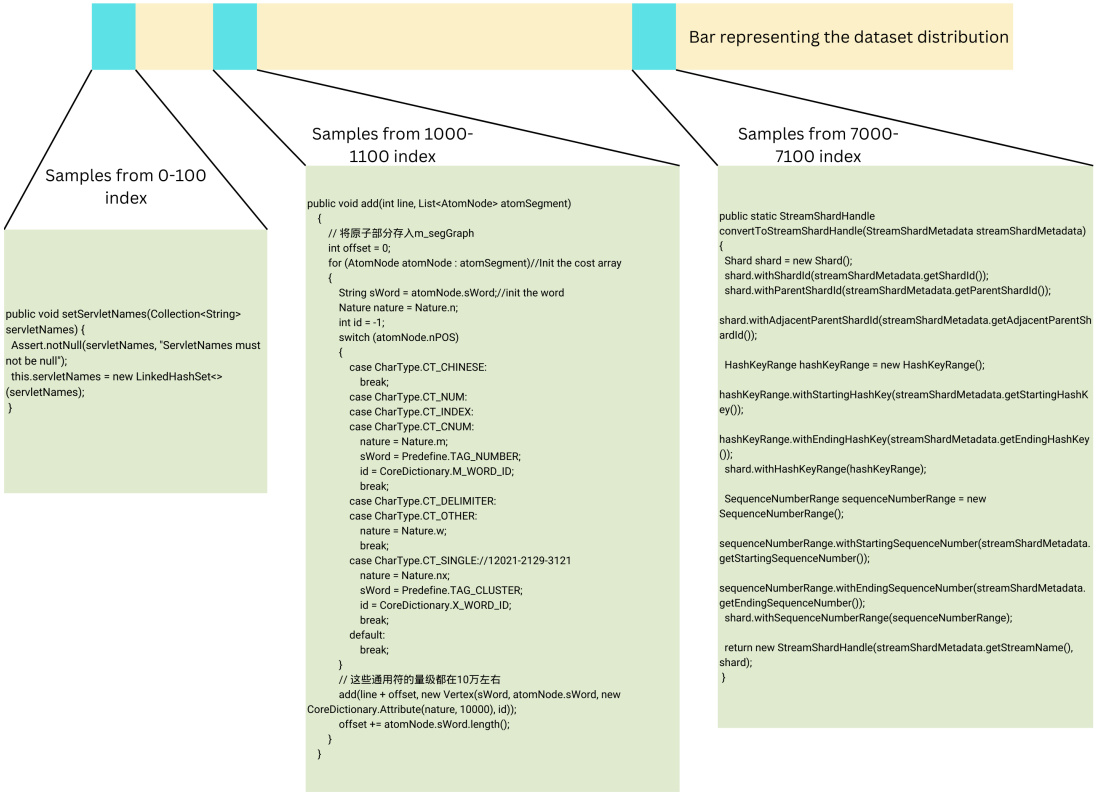


Figure A.43: Dataset samples from different index

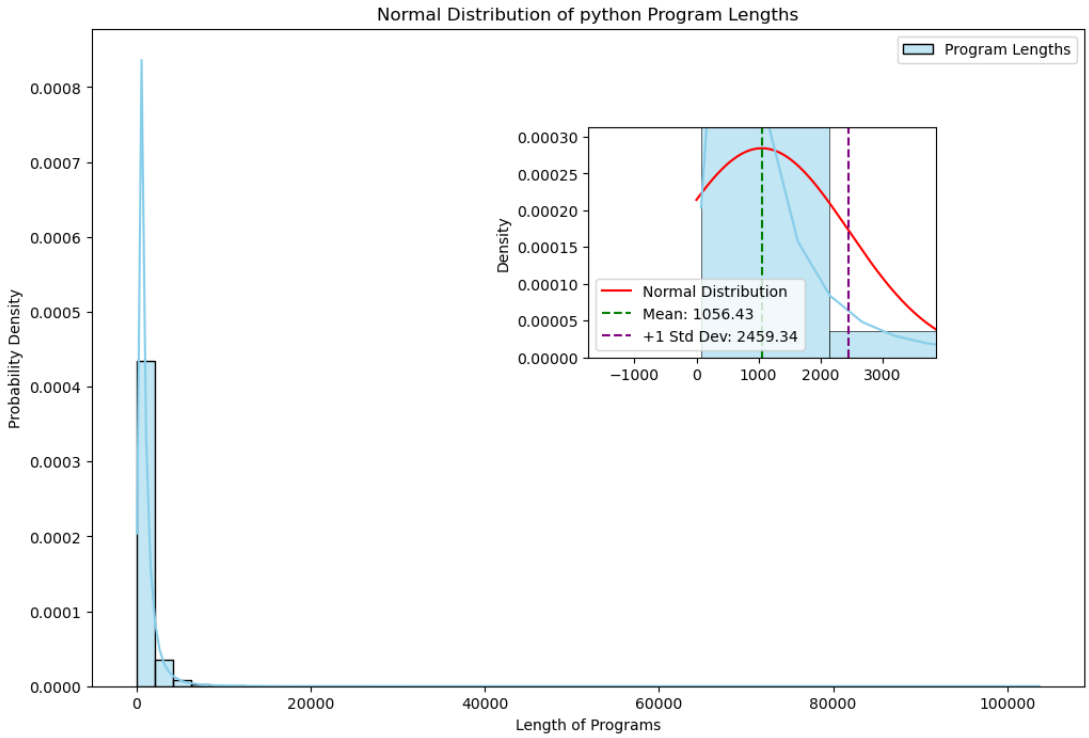


Figure A.44: Dataset Distribution for Python Language

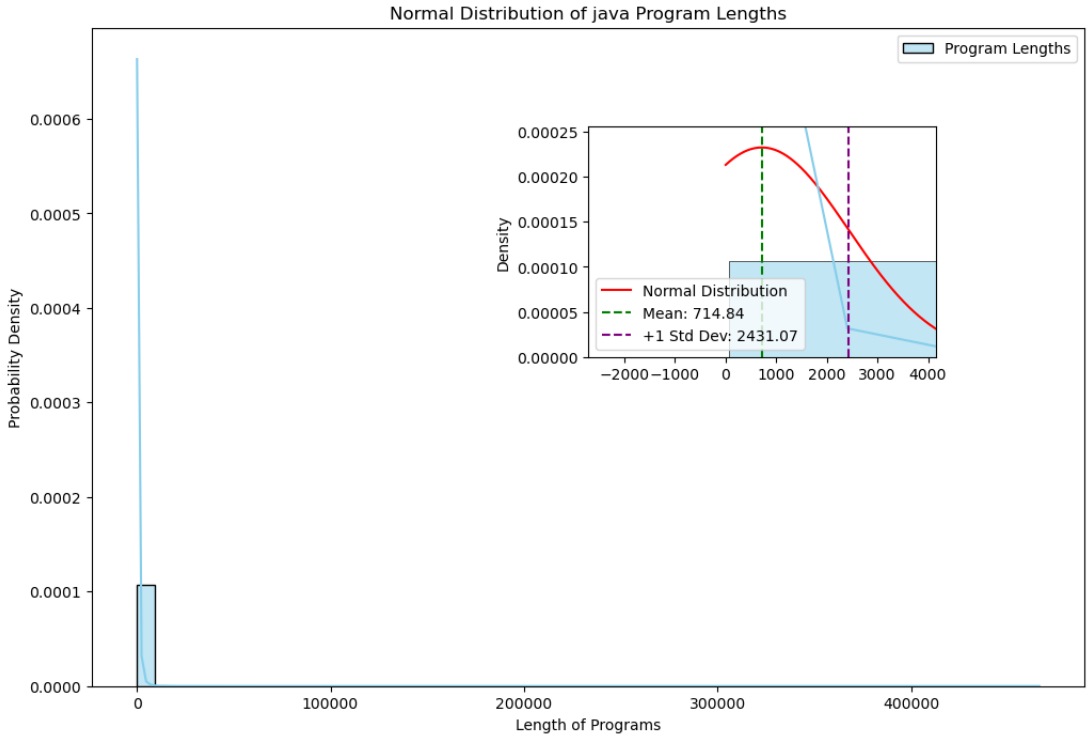


Figure A.45: Dataset Distribution for Java Language

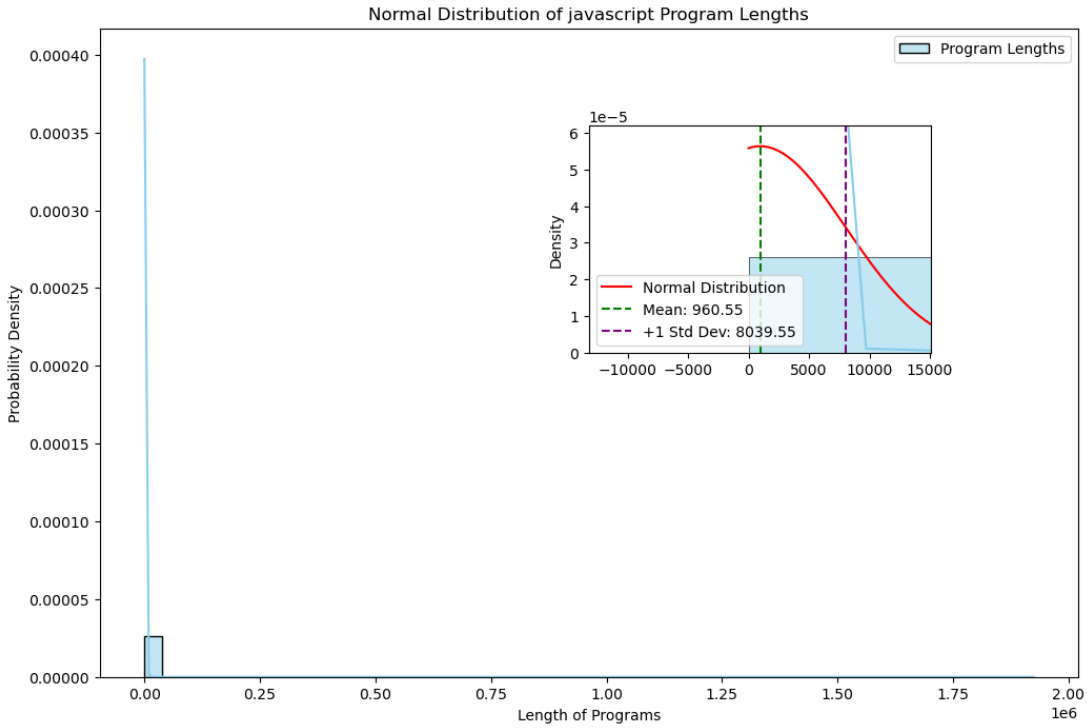


Figure A.46: Dataset Distribution for JavaScript Language

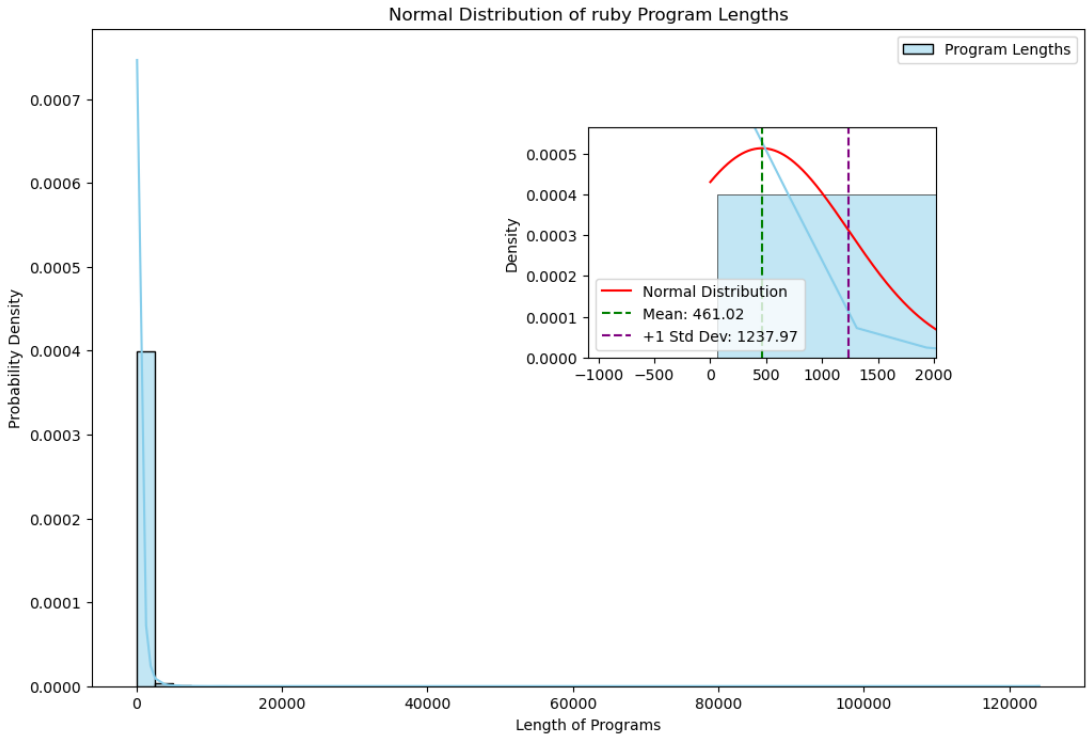


Figure A.47: Dataset Distribution for Ruby Language

## A.6 Experiment Setup

This section details the experimental setup used for conducting the memorization analysis across different code-based language models, programming languages, and learning paradigms. Hardware Configuration: All experiments were conducted on a high-performance computing cluster equipped with NVIDIA A100 GPUs. Each node in the cluster contained 8 A100 GPUs with 40GB of memory per GPU, 512GB of RAM, and 64 CPU cores ( utilizing The Eddie Cluster and the Cirrus Cluster from the University of Edinburgh Compute Resource). Software Environment:

Operating System: Ubuntu 20.04 LTS Python version: 3.11.7 PyTorch version: 2.2.2 Transformers library version: 4.42.3 CUDA version: 11.8

Model Specifications:

CodeBERT: 125 million parameters CodeGPT: 124 million parameters CodeT5: 220 million parameters

All models were initialized with their pre-trained weights as provided by their respective authors. Dataset: The CodeSearchNet dataset was used, comprising code samples from six programming languages. For this study, we focused on Python, Java, JavaScript, and Ruby. The dataset was preprocessed to ensure consistent formatting and to remove any samples exceeding the maximum sequence length of the models. Evaluation Metrics:

Exact Match Score: For masked language modeling tasks CodeBLEU: For code generation tasks Custom Privacy Metric: As described in Chapter 3, Section 3.3

Experiment Workflow:

Data Preprocessing: Code samples were tokenized and prepared for input to the models. Model Evaluation: Each model was evaluated on both masked language modeling and next token prediction tasks. Quantization: Models were quantized to 8-bit precision using PyTorch's dynamic quantization. Few-Shot Learning: Experiments were conducted with 0 to 3 examples for Few-Shot Learning scenarios. Prompt Tuning: Implemented using a gradient-based optimization approach over 50 epochs.

Computational Resources: The total computation time for all experiments includes time for model inference, quantization, Few-Shot Learning, and prompt tuning experiments across all programming languages and model architectures. Reproducibility: To ensure reproducibility, all random seeds were fixed at the beginning of each experiment. The codebase, including all scripts for data preprocessing, model evaluation, and analysis, has been version-controlled and is available in the accompanying GitHub

repository [https://github.com/vaikunth-coder27/MSc-Project]. This experimental setup was designed to provide a comprehensive and fair evaluation of memorization patterns across different model architectures, programming languages, and learning paradigms. The high-performance hardware and carefully controlled software environment ensured consistency and reliability in our results.

# Appendix B

# Participants' information sheet

No human participants was involved for this project.

# Appendix C

# Participants' consent form

No human participants was involved for this project.