

Approaching Memorization in Large Language Models

by

Xiaoyu Cheng

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2025

© Xiaoyu Cheng 2025

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Large Language Models (LLMs) risk memorizing and reproducing sensitive or proprietary information from their training data. In this thesis, we investigate the behavior and mitigation of memorization in LLMs by adopting a pipeline that combines membership inference and data extraction attacks, and we evaluate memorization across multiple models. Through systematic experiments, we analyze how memorization varies with model size, architecture, and content category. We observe memorization rates ranging from 42% to 64% across the investigated models, demonstrating that memorization remains a persistent issue, and that the existing memorization-revealing pipeline remains valid on these models. Certain content categories are more prone to memorization, and realistic usage scenarios can still trigger it. Finally, we explore knowledge distillation as a mitigation approach: distilling Llama3-8B reduces the extraction rate by approximately 20%, suggesting a viable mitigation option. This work contributes a novel dataset and a BLEU-based evaluation pipeline, providing practical insights for research on LLM memorization.

Acknowledgements

First and the foremost, I would like to express my deepest gratitude to my thesis supervisor, Professor Weiyi Shang, for his guidance, patience, and unconditional support throughout this project. His insightful advice and constant encouragement I received from the initial proposal to the final report were invaluable to the progression of this project, and to my development as well.

I also wish to thank Professor Pengyu Nie, who has introduced me to the key paper that inspired this project topic. Professor Nie continued to offer helpful feedback and mentorship throughout my research. That guidance significantly shaped the direction of my work and helped me to refine my ideas constantly.

I am very grateful to Professor Jialu Zhang, for kindly agreeing to be the reader of this thesis, and for committing to participate in my seminar. I sincerely appreciate his valuable guidance and feedback.

My appreciation goes to Dr. Kundi Yao, a postdoctoral researcher in our group, for his guidance in completing this project, and Professor Zishuo Ding for his valuable insights in refining the ideas behind this work.

Finally, I would like to thank all my lab mates from SENSE, for their unwavering support and friendship. Working alongside with such dedicated and supportive colleagues made the challenges from this work more manageable. I am truly fortunate to have been part of this team throughout my master's journey.

Table of Contents

Author’s Declaration	ii
Abstract	iii
Acknowledgements	iv
List of Figures	vii
List of Tables	viii
1 Introduction	1
2 Background	4
2.1 Memorization in LLM	4
2.1.1 The Complicated Nature of Memorization	4
2.1.2 The Problem Caused by Memorization	5
2.2 Revealing and Analyzing Memorization	7
2.2.1 Membership Inference Attack	7
2.2.2 Data Extraction Attack	7
2.2.3 Code Categorization	8
2.3 Attempts in Mitigation	9
2.3.1 Mitigating Memorization	9
2.3.2 Knowledge Distillation for Mitigation	10

3	Approach	11
3.1	Models Selected	11
3.2	Dataset Preparation	12
3.3	Membership Inference Attack	12
3.4	Data Extraction Attack	13
3.5	Evaluation Metrics and Categorization	13
3.6	Unintentional Attacks	14
3.7	Knowledge Distillation	14
4	Results and discussions	15
4.1	Memorization Revealing	15
4.2	Memorization Analysis	17
4.2.1	The Categorical Memorization	17
4.2.2	The Unintentional Memorization	20
4.3	Memorization Mitigation	22
4.4	Implications	25
5	Threats to Validity	27
5.1	Internal Threats	27
5.2	External Threats	27
5.3	Construct Threats	28
6	Conclusion	29
6.1	Conclusion	29
6.2	Future Work	29
	References	31

List of Figures

4.1	Memorization Rate for StarCoder2-3B	16
4.2	Memorization Rate for StarCoder2-7B	16
4.3	Memorization Rate for Llama3-8B	17
4.4	Memorization by Category for StarCoder2-3B	18
4.5	Memorization by Category for StarCoder2-7B	18
4.6	Memorization by Category for Llama3-8B	19
4.7	Unintentional Attack on StarCoder2-7B	20
4.8	Unintentional Attack on Llama3-8B	21
4.9	Membership Inference Attack Result for Llama3-8B	23
4.10	Membership Inference Attack Result for R1-distilled Llama3-8B	23
4.11	Extraction Attack for Memorization Rate LLama3-8B	24
4.12	Extraction Attack for Memorization Rate R1-distilled LLama3-8B	24

List of Tables

4.1	Memorization Performance on Large Language Models	17
4.2	Categorical Memorization Rates by Model	22

Chapter 1

Introduction

Large-scale, multi-modal foundation models have demonstrated strong performance across various tasks. These models are typically trained on public or proprietary datasets that consists of domain knowledge from different fields, in the form of text from various sources, including library database, internet based discussion forums, and Github-repositories [1]. With billions of parameters, state-of-the-art models demonstrate remarkable proficiency in different tasks such as natural language processing and code-related works, including code generation, summarization, and completion [2].

The progress in large language model (LLM) research has led to a widespread adoption of LLM-based applications in industry and academia. However, it has also raised serious concerns about models memorizing and repeating part of their training data, especially when that data contains sensitive personal information or proprietary code. Therefore, understanding and addressing memorization in LLMs has emerged as one of the most critical research priorities for safely deploying these models.

A major challenge in studying memorization is the lack of clear definitions and standardized metrics for model memorization. There is no universally accepted procedure to identify or quantify memorization in LLMs, particularly for code-focused LLMs. Recent work explores techniques to reveal memorization and potential data leakage, for example, a pipeline consists of membership inference attacks and data extraction attacks was utilized during the adoption of a data extraction “security game” [3]. Such approaches provide a foundation for assessing memorization, but they have only been applied to a small set of models and may not reflect the behavior of the latest LLMs. It remains unclear whether the existing pipeline is still effective, underscoring the need to extend memorization analysis to a broader range of state-of-the-art models.

Another challenge is how to quantify and interpret memorization in code-oriented language models. Code and natural language share certain common characteristics, but they are intrinsically distinct, as code is highly structured and more repetitive, therefore it is more predictable comparing to natural language text, and the predictability can often be utilized [4]. These characteristics could affect how an LLM memorizes code, for example, a code-generation model might memorize common code templates more easily. However, we still lack a clear understanding of what types of code snippets are more prone to memorization, or how to quantify the memorization risk. This gap points to the need for systematic investigation of memorization in code-focused LLMs.

Memorization can introduce various risks across multiple aspects, including privacy concerns, intellectual property issues, and model performance biases. These issues collectively make memorization a crucial problem for practitioners to solve. Researchers have explored various mitigation strategies, but approaches that have been proposed usually come with trade-offs between mitigation effectiveness and model performance. This issue calls for methods that are both efficient and balanced.

Given the above challenges and gaps, this thesis aims to deepen the understanding of memorization in large language models, and to explore techniques for its evaluation and mitigation.

RQ1: Does the existing memorization-evaluation pipeline remains valid on state-of-the-art LLMs? Prior findings were evaluated on models that are less commonly used [3]. We adopt the existing attack-based pipeline to measure memorization rates in recent models, including the StarCoder2 and Llama3 families, and assess whether the approach is still effective.

RQ2: How does attribute of code like categories affects memorization, and how easily does realistic usage scenarios trigger memorization? Code offers attributes that can be extracted for analysis, for instance, practitioners have categorized code snippet into different categories. Therefore, we analyze code snippets by categories to see which are more prone to memorization. We also propose unintentional attacks, to simulate realistic usage scenarios, and to evaluate how memorization affects day-to-day use.

RQ3: Can we find a cost-effective mitigation approach? We explore preliminary mitigation via knowledge distillation, aiming to reduce memorization without overly compromising performance.

The contribution of this work includes the followings:

- Validation on recent models: we adapt a prior memorization-detection pipeline for

LLMs and apply it to recently released models (StarCoder2 and Llama3 families), validating the pipeline and providing evidence of memorization in current LLMs.

- Categorical analysis: we categorize code segments by functional purposes and use these categories to quantify memorization. For the “code” category, we perform simulations of realistic usage scenarios, and evaluate the memorization risk associated with it.
- Preliminary mitigation: we investigate knowledge distillation as a mitigation strategy and report preliminary findings on reducing memorization with limited performance loss.

Overall, this work addresses key gaps in the current understanding of memorization in large language models. By validating memorization revealing pipelines, analyzing the memorization behavior, and investigating practical mitigation approaches, we aim to provide useful insights for researchers and practitioners, and shed some light on memorization in LLMs.

Chapter 2

Background

2.1 Memorization in LLM

2.1.1 The Complicated Nature of Memorization

Generative models have shown an unexpected tendency to memorize segments from their training data verbatim, rather than generalize from them. This behavior came to light when practitioners discovered that unique sequences can be repeated word-for-word by the model. For example, Carlini et al. demonstrated that sequences such as synthetic credit card numbers injected into training data, could later be extracted from a model, concluding that unintended memorization is persistent [5]. Memorization also appears arbitrary, for example, memorization can be considered as a result of overfitting as practitioners have suggested, but in the meantime, researchers have also suggested that even training with a single epoch can contribute to memorization [6]. Moreover, studies have reported that memorization is common across architectures and training strategies, and that larger models could be more prone to memorization [5]. A recent study on code generation also highlights the severity of unintended memorization: among 20,000 outputs, more than 40,125 code snippets were directly memorized from the training set [7]. These observations have made clear that unintended memorization is a crucial problem for practitioners to investigate. However, there still lack a clear indication of the behavior pattern of memorization, which has made memorization more challenging and concerning.

2.1.2 The Problem Caused by Memorization

Memorization in large language models tend to pose several serious risks. In this section, we discuss some of the major problems caused by memorization, including concerns with privacy leakage issues, risks associated with intellectual property violations, and model performance bias that reduces generalization. Each of these concerns illustrates why memorization must be carefully managed in order to safely utilize LLMs in production.

Privacy and data leakage is one of the most critical issues associated with LLM memorization [8]. Studies have shown that large language models can be exploited to extract segments of their training data through memorization, the extracted data can include highly sensitive personal identifiable information. Carlini et al. demonstrated that, through data extraction attacks on GPT-2, the model could reproduce content such as individuals' names, phone numbers, and email address, as well as proprietary information, such as source code and unique identifiers (UIDs) [6]. In another study, Carlini et al. also noticed that a model can reproduce an injected synthetic credit card number [5].

Such leakage of sensitive information poses severe risks: malicious users could use the extracted personal data or identifier for identity theft and fraud, and organizations deploying LLMs might unintentionally release personal information without consent [9]. More worryingly, attackers can bypass a model's defense mechanisms, and retrieve data even if the model is designed to avoid verbatim outputs, as attackers can craft alternate prompts or use partial information to recover sensitive information, and simply installing naive filters could only provide a false sense of security [10]. Therefore, unintentional memorization remains a crucial problem, it enables adversaries to query the model and returns confidential training data, which should have remain non-disclosed, and it significantly limits the safety deployment of LLMs, especially ones that might contain personal or proprietary information.

Memorization also leads to risks towards the use of intellectual property, and creates potential copyright violations when model repeats copyrighted or licensed material from its training set, as LLMs tend to overlook license information. LLM trained on data from internet may unintentionally memorize and reproduce copyrighted text and open-source code that are governed by specific licenses, causing potential legal liabilities and ethical problems, especially in commercial applications [7]. Developers have observed that code generation models occasionally output recognizable code segments from open-source repositories without referencing the license, and models like Copilot can generate copyrighted code in sizable segments with no attribution or license [11]. This creates a scenario where users might unknowingly include unlicensed code in their projects. Copyright concern

appear common, findings suggested that even the top-performing LLMs produce a considerable amount of code that is significantly similar to open-source implementations.

The legal and ethical issue resulted from such behavior can be serious. For example, using a piece of code without properly acknowledging the license can lead to copyright claims, potentially resulting in lawsuits or even changes to the ownership and forced open-sourcing a proprietary project. More importantly, this could lead to trusting issue and discouraging developers from using LLMs [12]. Therefore, the possibility of unintentionally producing licensed or copyrighted materials means memorization must be controlled and regulated to avoid plagiarism and intellectual property violations.

Memorization can lead to biased behaviors with overfitting and reduced generalization, as a model that over-fits its training data may simply repeat its training examples, which makes the model less capable of generalize on new, unseen tasks [13]. The lack of generalization significantly degrades the quality and reliability of the output on novel tasks, and the model might prioritize its output towards memorized responses instead of results with better correctness, which reduces the reliability of LLMs on critical tasks.

Another issue with biased behavior, is that LLM tends to inherit the stereotypes presented in the training data. For example, an LLM may reinforce stereotypes on gender and ethnic groups, simply because of the existence of biased material in the training text [14]. Such behavior leads to a skewed output distribution, which reduces the model’s objectivity and reliability, and might cause further problems including racism or sexism in the generated content.

These biased behaviors pose a great threat to LLM performance. Models lack of generalization can be considered under-performed and lacking robustness, and models outputting skewed results failed in maintaining fairness [15].

In conclusion, these concerns make unintended memorization a challenging problem to solve, it can violate user privacy, create intellectual property conflicts, and degrade the model’s performance through overfitting and creates biases. These issues have encouraged researchers to focus on understanding memorization, and developing effective approaches to mitigate it. Addressing memorization is critical for practitioners to safely use LLMs in a responsible manner.

2.2 Revealing and Analyzing Memorization

2.2.1 Membership Inference Attack

To investigate memorization, it is required to have knowledge on a model’s training data. However, most models do not disclose their training dataset. To determine if certain data is contained within the training set, a common approach is through membership inference attack (MIA).

Originally introduced by Shokri et al. as a game-based theoretic concept, a membership inference attack is described as a security game: an adversary wins if it can correctly distinguish a sample that the model has seen during training or not [16]. MIA is based on the assumption that the model behaves differently on seen data and brand new data. Practitioners believe that models tend to overfit to samples from training set. As a result, models might perform better with a higher confidence scores for seen data [17]. Therefore, the gap in performance provides a promising approach to identify seen data [18].

For LLMs, especially in code completion tasks, membership inference attack can be implemented by prompting the model to generate an output for a given input, then measuring how closely the output matches the original segment [19]. If the generated data matches the original data closely, we can infer the data was likely seen during training. Exact match between the model’s generated content and the original sequence can be considered as a strong indicator of existence in the training set. However, even partial matches can be informative, as BLEU score is used to evaluate fuzzy matches [3]. A high BLEU score suggests the model’s output captures a considerable amount of the original content, and indicates the model had likely seen it during the training process. Whereas a low BLEU score could suggest the tested sequence is new.

Membership inference attacks were proven to be an effective and practical approach auditing and identifying whether a sample is in the training set. However, it does not reveal if the sample is memorized and reproducible. To demonstrate memorization leakage, we need to utilize a more sufficient approach, which leads to data extraction attacks.

2.2.2 Data Extraction Attack

A data extraction attack is a stronger form of attack built upon membership inference attack, aiming to recovering memorized data and quantify a model’s memorization behavior. Once a membership inference attack result suggests that a model has likely seen a specific

data sample, a data extraction attack is performed on the seen sample, and the goal is to reconstruct the sample by querying the model. Successful extraction is considered as an evidence of memorization, as the model is induced to reveal verbatim content. Together, membership inference and data extraction attack form a sequential pipeline that efficiently reveals memorization [3].

There are two common types of data extraction attacks that are often being discussed, the unguided attacks and the guided attacks [20]. The major difference is that in unguided attacks, the adversary does not maintain the knowledge of the specific data to extract, and attempts to recover training sample from the dataset. Whereas in guided attacks, the adversary is provided with a pre-fix that is the first-half of the target sequence, and aim to reconstruct the corresponding suffix that is the second half of the target sequence. Guided attacks pose a greater security risk, as sensitive information can be revealed using this approach, for instance, feeding an LLM the beginning of a confidential sequence it saw in training might cause it to output the rest of the sequence.

In our work, we focus primarily on guided extraction attack, as it offers a more deterministic approach to evaluate memorization. Similarly, this approach uses exact match and BLEU score to represent the likelihood of memorization. This approach has revealed the fact that LLM can memorized unique sequences from the training verbatim and reproduce it, thus causing leakage and privacy breach.

Taken together, MIA and extraction attack form a promising pipeline, with which it can successfully transform an abstract concept of memorization into a measurable object. This pipeline provides a systematic approach and have been proven successful by previous studies [3]. However, the previous work does not contain newer models, so we aim to validate whether the pipeline still remain effective on state-of-the-art LLMs.

2.2.3 Code Categorization

Our investigation focuses on code completion tasks, meaning that the subject of this thesis are code-related snippets. Though code and natural language share various similarities, code is intrinsically different from natural language, and tend to be more predictable [4]. Al-Kaswan et al. categorized code snippets into five different categories, and reported that certain categories can have different implications towards memorization [3]. We conduct similar categorization procedures, aiming to validate if the previous findings still hold, and to help us gain better understanding towards the memorization behavior.

Focusing on code completion tasks, we consider how ordinary queries can unintentionally trigger memorization. For example, developers often ask LLM to fill in function body

given only its signature, and it has been reported that even prompts without a rigid structure of adversary attacks, can cause the model to output memorized sequences from its training verbatim [21]. Therefore, we would like to investigate whether memorization can be triggered, if we use a method header combined with description as input. We characterize such behaviors "unintentional attacks", because the user's intent is benign, but the model spontaneously reveal memorized content. Studying the unintentional attack can help us better understand memorization in practice.

2.3 Attempts in Mitigation

2.3.1 Mitigating Memorization

A variety of mitigation strategies have been purposed and explored to address the issues caused by memorization. One straightforward approach is dataset de-duplication. Lee et al. found that language corpora usually contain a significant amount of near-duplicate sequences, and deduplicating could effectively improve the training process, as well as reducing the memorization tendencies [22]. However, deduplication does not eliminate memorization completely, models can still memorize unique sequences even if they only appeared once [5]. Moreover, code is naturally repetitive, which makes it not practical to de-duplicate training set [4], and deduplicating could reduce the diversity of data.

Another mitigation approach is differential privacy. Differential privacy provides a framework, within which the training data can extend its content with noisy, random, non-sensitive data. This approach guarantees that whether an individual's datapoint is present or absent, it only poses a negligible influence on the outcome. However, the added noise often causes the privacy-performance trade-off, models trained with differential privacy often demonstrate degraded performance on downstream tasks [23].

Recently, researchers proposed a mitigation method called *goldfish loss*, that can be used to tweak the training process, during which an arbitrarily selected subsection of tokens are excluded from the loss computation process, and the selected tokens are dropped and are not being memorized by the model [24]. By never learning those dropped tokens, the model is prevented from reproducing the entire sequence. By applying goldfish loss during Llama-2 training process, researchers have observed significant reduction in memorization, with little impact on performance [24]. However, this approach has its downside too, implementing goldfish loss means the model must continuously decide which tokens to drop during training process, this frequent modification could incur significant cost in training performance.

Each of these techniques involves trade-offs, stronger mitigation approaches often lead to higher costs or performance limitations. Given that no current approach can eliminate memorization entirely. Practitioners aim to find the balance between mitigating efficiency and performance degradation or cost [25]. In practice, the goal is to reduce risk to an acceptable level while retaining the model’s usefulness.

2.3.2 Knowledge Distillation for Mitigation

Knowledge distillation has emerged as a technique that is increasingly used for large language model [26]. In knowledge distillation, a large teacher model’s behavior is used as an example to teach a smaller student model, as a result, the student model is trained to optimize and mimic the teacher’s behavior, and apply it to its own inputs. Therefore, the knowledge and behavior pattern from the teacher model can be preserved and applied [27]. While originally designed for model compression and simplification, knowledge distillation have drawn interests for its potential in mitigating memorization [28]. The intuition is that student learns more generalized behavior, without memorizing the specific training example, enabling knowledge distillation to act as a privacy-oriented mitigation.

Knowledge distillation achieves two goals at once, with the improved generalization ability and the promising performance on downstream tasks, making knowledge distillation holds the potential for being a efficient mitigation method. However, there is no silver bullet, the effectiveness of knowledge distillation on LLM memorization still needs further investigation.

Chapter 3

Approach

We constructed a research pipeline aiming to answer the proposed research questions. The pipeline consists of a two-stage adversary attacks, inspired by recent work on LLM memorization [3]. First, we construct a custom code dataset, and identify any samples that were likely presented in a model’s training set using membership inference attack. Next, we perform guided data extraction attack on those identified candidates to recover memorized content. After the adversary attacks, the result is then categorized and evaluated. Continuing the categorization result, unintentional attacks are then performed on the code logic category to better understand memorization in the scenario of real-life practices. Finally, we compare the memorization between a model and its distilled version, to assess the impact from knowledge distillation.

3.1 Models Selected

There are four different open source LLMs used in this project, two from the StarCoder2 family and two Llama3 variants.

StarCoder2 targets code-related tasks. We have incorporated both the three billion-parameter variant (StarCoder2-3B) and the seven billion-parameter variant (StarCoder2-7B), which share the same architecture and training corpus, and only differs in parameter size. The StarCoder families are trained on a carefully curated set called “The Stack V2”, which is a corpus that consists of source-code [29]. By comparing the performance of these two variants side by side, we will be able to tell whether a 133% increase in size leads to more verbatim memorization.

We also investigated if Llama models revealed similar behavior. Llama3 is a family of open source models introduced by Meta [30]. We chose the variant with eight billion-parameters (Llama3-8B), which is comparable in size with the previously selected StarCoder2-7B. However, because the families are different intrinsically in their intrinsic properties, we are conducting this part of experiment mainly to determine if memorization is primarily driven by model size or by the nature of the model.

The last model is the DeepSeek-R1-distilled-Llama-8B. Our goal is to gain insights about how knowledge distillation affects memorization. DeepSeek-R1 is an open-source reasoning model made available by DeepSeek-AI, it is used as a teacher model, whose behavior patterns are distilled into Llama3-8B the student model [31]. By using the DeepSeek-R1-distilled-Llama-8B, we can assert the hypothesis that knowledge distillation can reduce memorization.

3.2 Dataset Preparation

Instead of relying on existing benchmarks, we constructed our own dataset to minimize evaluation and performance bias.

We leveraged the Github-repos dataset that is hosted on BigQuery [32], and curated a Python code dataset.

This dataset includes file-level metadata and content from a large corpus of open-source repositories. We tokenized files using the default tokenizer of each target model, and we randomly extracted 150-token sequences. To enable fine-grained prompting, each 150-token sequence is segmented into three parts with equal lengths, which we refer to as the pre-prefix, prefix, and suffix. The pre-prefix and prefix are the beginning of a potentially memorized snippet, and the suffix is the ground-truth that we aim to refer to.

We collected over 30,000 random python code files. This diverse corpus reduces bias and ensures the diversity of the content.

3.3 Membership Inference Attack

After constructing the dataset, the goal is to find samples that exist in the training corpus. We employ the membership inference attack technique and adopted the approach used by Al-Kasawn et al., in which the model is queried with the first part of each sequence, and its output is compared with the remaining part [3].

In our dataset, a segment that is 150 tokens long is split into three parts with equal lengths: pre-prefix, prefix, and suffix. We prompt the model with the concatenation of the first two parts, and we observe whether the output matches the known suffix, and only sequences that successfully trigger the model to produce the correct continuation are considered “seen” during training, and can be used as candidates for memorization analysis. By using a relatively long prompt, we maximize the confidence of model identifying the seen context correctly.

The Stack V2 maintains a metadata dataset indicating files that have been used for training, but it is not as reliable as experimentally revealed results, because the Stack V2 is de-duplicated, and does not include repository that were forked or cloned, meaning that a file can still exist in the training set, even the original repository that the file belongs to was not included in the training set. Also the content repository is constantly changing, therefore using the repository snapshot as a reference may not provide the most accurate result.

3.4 Data Extraction Attack

For each candidate snippet identified by membership inference attack, we conduct a guided data extraction attack to evaluate memorization. In this stage, the model is prompted only with the 50-token prefix (without the pre-prefix). We generate the model’s completion and compare it with the original suffix. If the output sequence closely matches the ground-truth suffix, we then consider the attack successful and interpret it as an evidence of memorization. By limiting the input to only the prefix, we can test the model’s ability to reproduce training data with minimal context, which is a strong indicator of memorization. This strategy was used by Al-Kaswan et al., and has proven to be an efficient memorization-revealing method [3].

3.5 Evaluation Metrics and Categorization

After data extraction attacks, the results are evaluated. Exact match is a strict criteria that may not count fuzzy matches that is nearly identical. Following Al-Kaswan et al., we also compute the 4-gram BLEU score, which captures partial memorization cases [3].

We then divide test segments into categories for analysis. Prior work reported that different categories memorize differently on models like the CodeGen-family [3]. We would

like to validate if the previous findings remain valid on newer models included in our study. There are five categories previously proposed, including “code” for code logic, “testing” for testing code, “license” represents license information, “docs” for documentation, and “dicts” for dictionary and other data carriers [3]. Each memorized sample is tagged with one of these categories according to the file context. This allows us to investigate whether certain types of content are more prone to memorization, and we can validate if previous findings still hold for the subject models and dataset. This categorical evaluation provides deeper insights into what kinds of information each model memorizes more.

3.6 Unintentional Attacks

To simulate a realistic code-completion use case, we prompt each model with the beginning portion of a code file, and examine its output. Instead of using a fixed-length 50-token prefix, we use a more natural prompt length and content, for example, the function signature and the doc-string of a file. We then let the model continue generating code, but limiting the output length to 50 tokens, ensuring a efficient comparison. The inputs are actual functions beginnings, that a developer might naturally use without malicious intent. The tests were ran on StarCoder2-7B, and Llama3-8B, using a dataset that only contains files from the code category from our categorization results.

3.7 Knowledge Distillation

In addition to testing base models, we investigate how knowledge distillation affects memorization in LLMs, and incorporated a distilled model into our experiments to test this hypothesis in practice. Specifically, we evaluate the Llama3-8B model via knowledge distillation by DeepSeek-R1.

Our goal is to probe whether knowledge distillation can reduce memorization. We evaluate the distilled Llama3-8B model using the exact same memorization revealing pipeline, and by controlling all parameters, the only variable is the knowledge distillation process. The result is then compared against the result from the original Llama3-8B.

Chapter 4

Results and discussions

4.1 Memorization Revealing

Research Question Summary 1

Memorization persists in newer models, and the existing memorization-revealing pipeline remains valid.

The memorization results for the three models that we investigated are presented in [Figure 4.1](#), [Figure 4.2](#), and [Figure 4.11](#). These figures illustrated the distribution of a model’s memorization rate for code samples.

All models investigated here reveal memorization, with a rate ranging from 42.24% for the Llama3-8B model, to 62.00% and 64.16% for the StarCoder2-7B, 3B model, respectively.

For the two variants from the StarCoder2 family, the increase in model size did not translate to a higher memorization rate, instead, the larger model memorized slightly less as suggested both by exact match and average BLEU score. This observation is different from what previous findings revealed, and one possible explanation is that at this scale of parameter size, the effect from size on memorization is negligible. The similar sized Llama3 model revealed a lower memorization rate comparing to the StarCoder2-7B, which might because of the training set differences, as StarCoder2 is exclusively trained on code snippets, and Llama3 has more general objectives. Overall, our experiments verify that the memorization-detection pipeline remains effective for newer models, indicating that even recent LLMs can memorize portions of their training code corpus. This finding reinforces

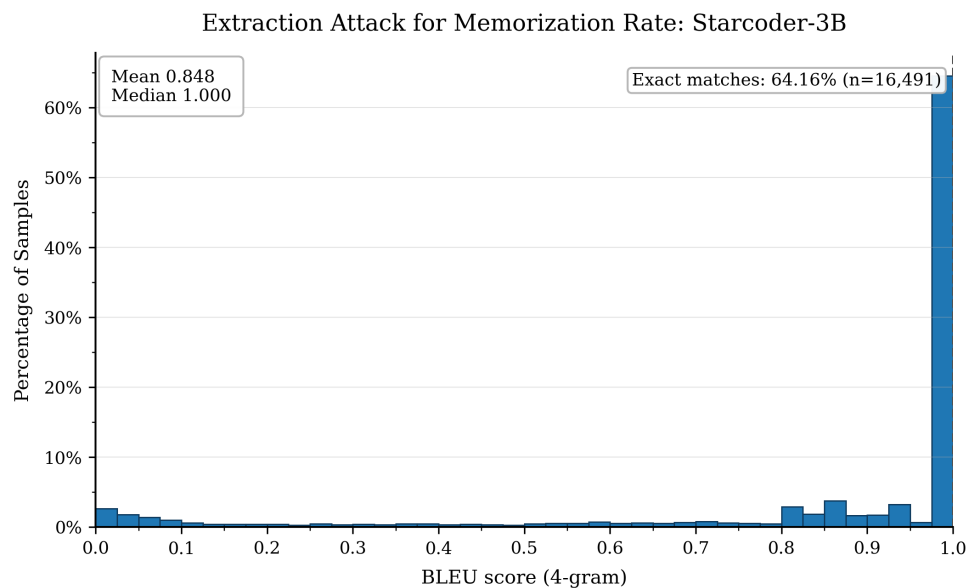


Figure 4.1: Memorization Rate for StarCoder2-3B

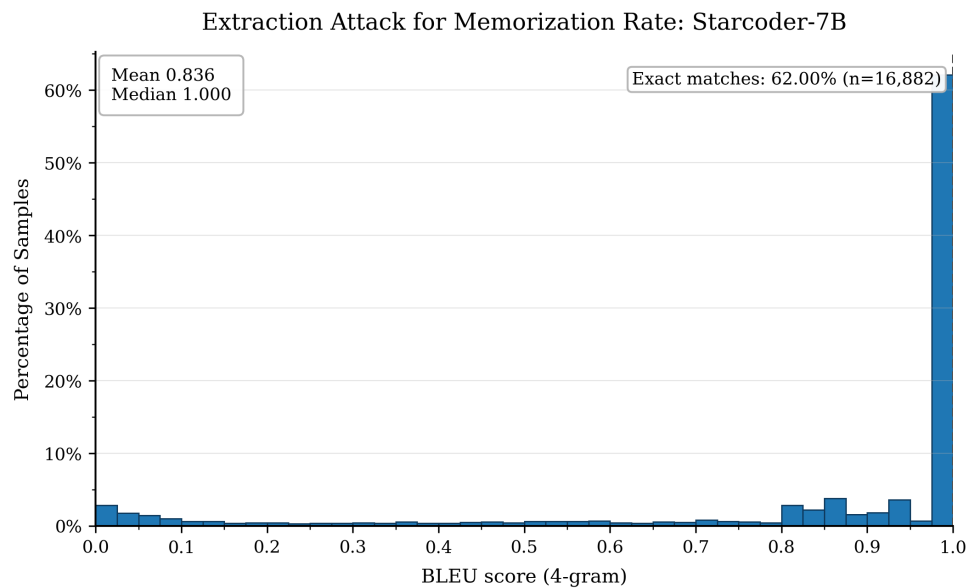


Figure 4.2: Memorization Rate for StarCoder2-7B

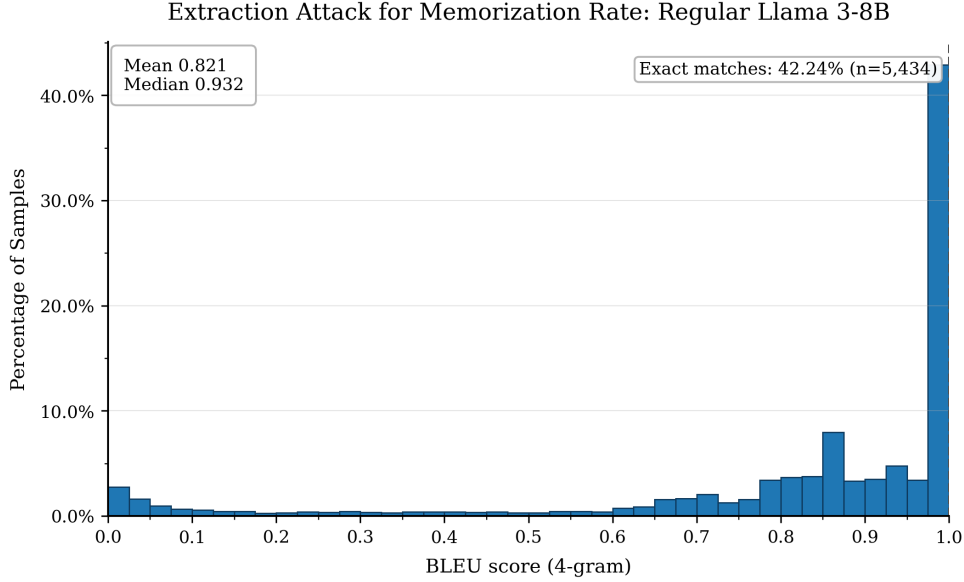


Figure 4.3: Memorization Rate for Llama3-8B

concerns raised in prior work, and indicates that unintended memorization is an intrinsic behavior of LLMs which still requires further investigation.

Table 4.1: Memorization Performance on Large Language Models

Model	Memorization rate	
	EM (%)	BLEU-4
StarCoder2-3B	64.16	0.848
StarCoder2-7B	62.00	0.836
LLaMA 3-8B	42.24	0.821

4.2 Memorization Analysis

4.2.1 The Categorical Memorization

We next investigate whether certain types of code are more prone to memorization, and what attributes might drive these differences, by analyzing the result at a finer granularity.

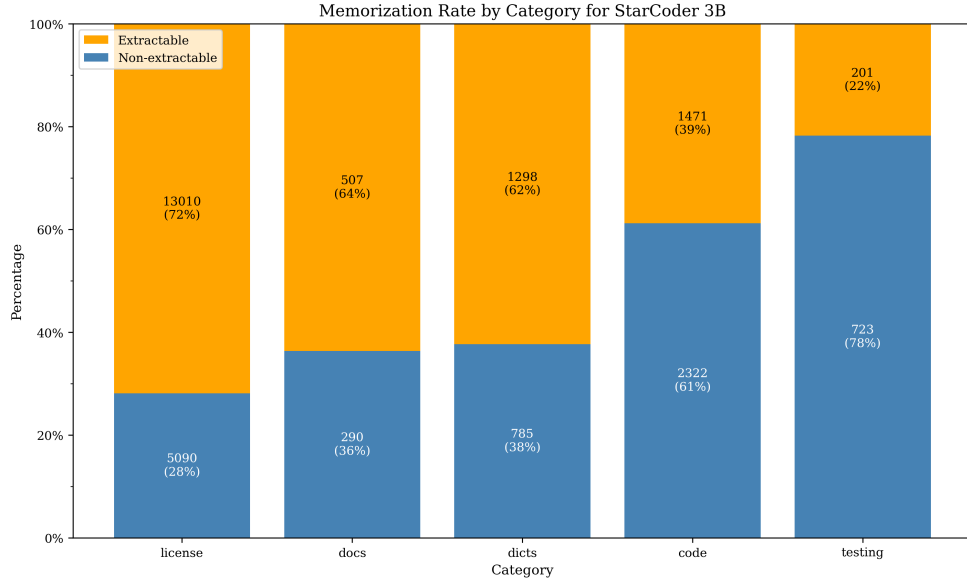


Figure 4.4: Memorization by Category for StarCoder2-3B

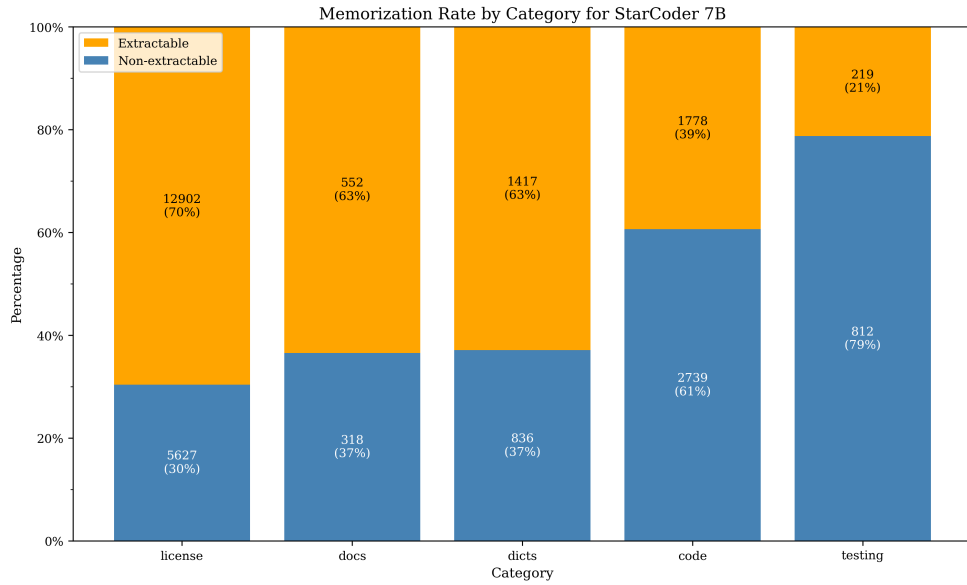


Figure 4.5: Memorization by Category for StarCoder2-7B

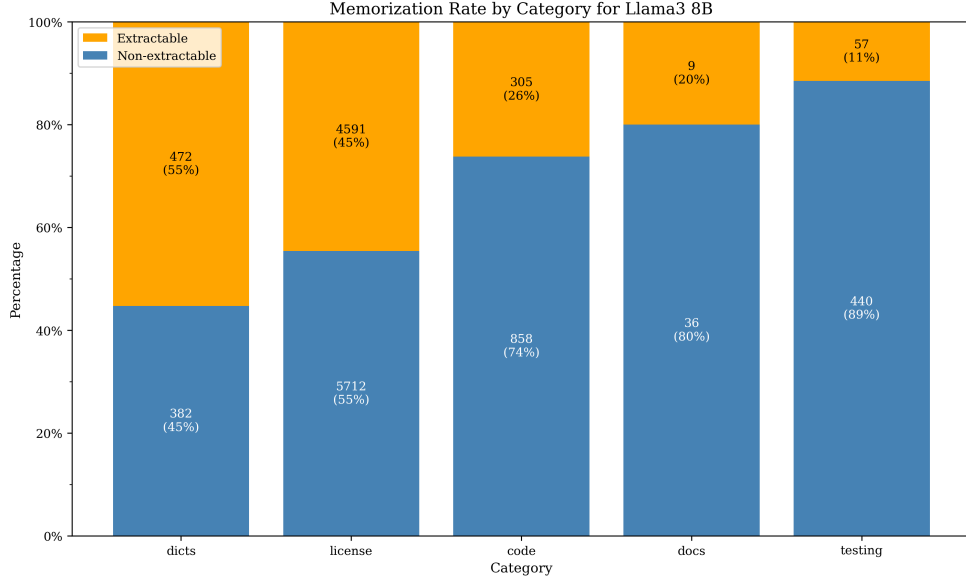


Figure 4.6: Memorization by Category for Llama3-8B

Prior work on code LLMs found that memorization favors certain types of code content, for example, Al-Kaswan et al. categorized code files and observed that license test and data-containing files tend to be memorized at higher rates than other categories, while testing is the least memorized [3]. To better investigate this, we grouped the extracted code by the same categories and measured memorization rates for each category on selected models, then we analyze the normalized results.

As discussed in previous sections, five categories were used, and the result has revealed clear distinctions. For both StarCoder variants, results are demonstrated in Figure 4.4 and Figure 4.5 license information is the most memorized category: over 70% of license containing snippets in the test set were revealed to be memorized by the model. Following licenses, snippets that contain documentation and data carriers like dictionaries are revealed with high memorization rates, with well above 60% of the related snippets being memorized. Code logic snippets were memorized at a moderate rate at around 39%, and testing code was memorized at the lowest rate of 21% to 22% by the StarCoder2 models. For the Llama3 model, the result is shown in Figure 4.6, and has reported a slight different distribution, but it also showed a strong tendency to memorize license text and data carriers, but revealed a lower memorization rate on documentation and code logic comparing to StarCoder models.

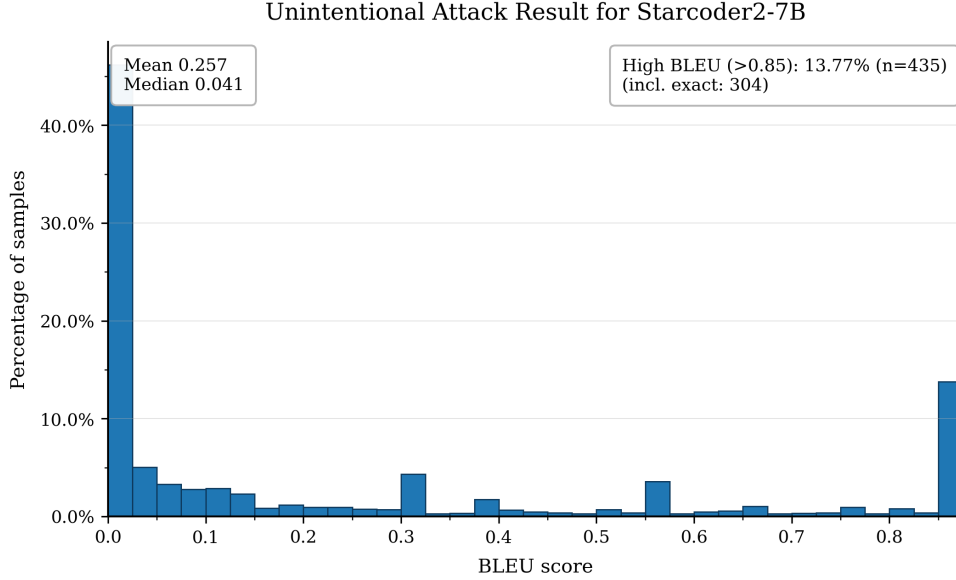


Figure 4.7: Unintentional Attack on StarCoder2-7B

Regardless of some of the variations between model families, the result still aligns with prior expectations: license files are being memorized the most across all models, whereas testing snippets are being the least memorized category. The result aligns with the intuition, as license texts are usually highly repetitive, and exist in many repositories. Therefore, the model sees them frequently in the training set and can recall their verbatim easily.

In contrast, testing code are usually less repetitive and are typically designed specifically for function units. Therefore, the model sees less of the repetitive testing code, and making it harder to memorize the exact verbatim. Overall, our findings align with the previously reported result, that memorization is not uniform across different categories of snippets, with repetitive, boilerplate content being susceptible to memorization, while complex, unique sequences are relatively less prone.

4.2.2 The Unintentional Memorization

Beyond snippet categories, we also investigated the realistic usage scenario of memorization. According to results being reported in [Table 4.2](#), code-logic snippets are being memorized at a range from 26.23% to 42.77%. However, these rates were measured under strict

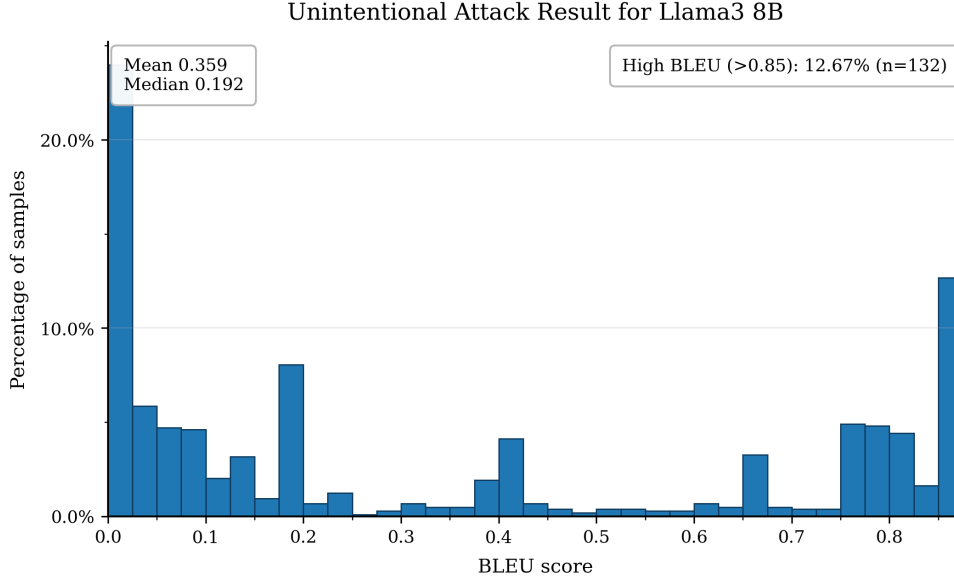


Figure 4.8: Unintentional Attack on Llama3-8B

attack conditions, with a fixed length input, which rarely holds in practice. Especially for code completion tasks, user typically provide a descriptive first half, including the function signature, and comments, and let the model generate the function body. To validate whether the realistic usage scenarios pose significant memorization threats, we evaluated completions by inputting the concatenation of a function signature and its subsequent comments.

Results show that when being prompted unintentionally, the model has a moderate chance reproducing the original training data. For the StarCoder2-7B model, the result is presented in Figure 4.7, around 13.8% (435 out of 3158) generated completions contained samples that are close to exact match with a BLEU score higher than 0.85, and for Llama3-8B, as shown in Figure 4.8 around 12.7% (132 out of 1042) completions met the same criteria. These results indicate that a significant portion of the training data can be revealed or likely revealed in realistic usage scenarios. Suggesting that tools like AI coding assistant may generate licensed code or other proprietary snippets without proper references, even when not being purposely instructed to perform malicious attacks.

Table 4.2: Categorical Memorization Rates by Model

Model	Memorization rate (%)				
	license	docs	dicts	code	testing
StarCoder2-3B	71.88	63.61	62.31	38.78	21.75
StarCoder2-7B	69.63	63.45	62.89	42.77	21.24
LLaMA 3-8B	44.56	20.00	55.27	26.23	11.47

Research Question Summary 2

Certain categories, such as license text, are more prone to memorization than others. Realistic usage scenarios can trigger unintentional memorization in the code category.

4.3 Memorization Mitigation

Finally, we explored whether knowledge distillation can mitigate memorization in LLMs. To test these, we chose Llama3-8B as the original model, and the DeepSeek-R1-distilled-Llama3-8B as the distilled version, and we ran the same evaluation pipeline. The results suggest that distillation can significantly mitigate memorization. The distilled Llama3 model shows a notably lower rate of memorization in the output result comparing to the original model.

In membership inference attacks, comparing to the result from the regular model in Figure 4.9, the distilled result showing in Figure 4.10 suggests that it generated way less positive outputs than the original model, indicating that data points present in the training set no longer get identified through the membership revealing attacks. More importantly, the extraction attack success rate has dropped from 42.24% to 34.17% from the original model to the distilled version, as indicated in Figure 4.11 and Figure 4.12. Meaning that the distilled version “forgot” around 19% of the memorized content. The reduction is significant given that both models are trained on the same dataset, suggesting that the distillation process itself can aid reducing memorization. Our finding is consistent with the intuition that knowledge distillation forces the student to prioritize general learning patterns instead of repeating specific training examples.

The result appears promising, as knowledge distillation seems to be a relatively cost effective technique that can be easily deployed. Especially comparing to traditional defenses

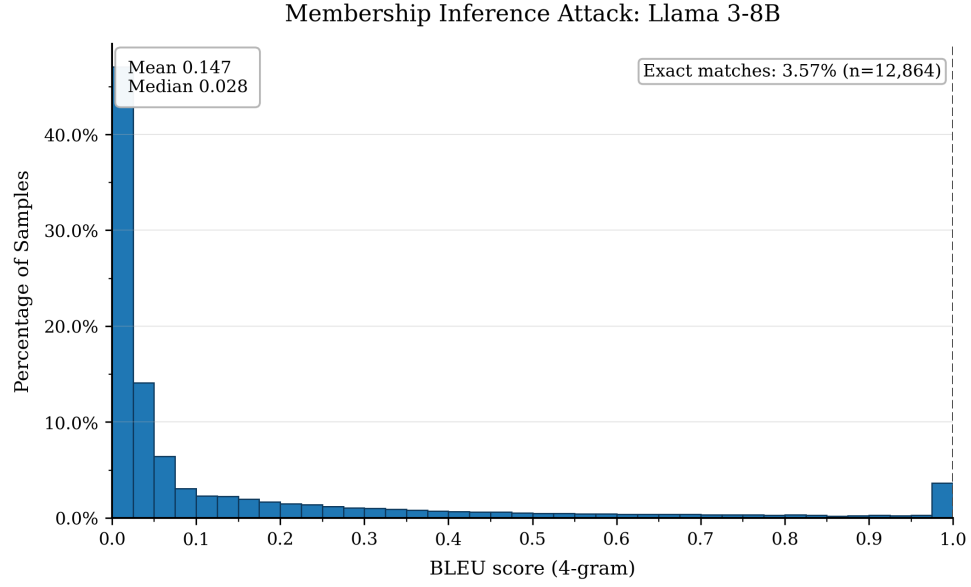


Figure 4.9: Membership Inference Attack Result for Llama3-8B

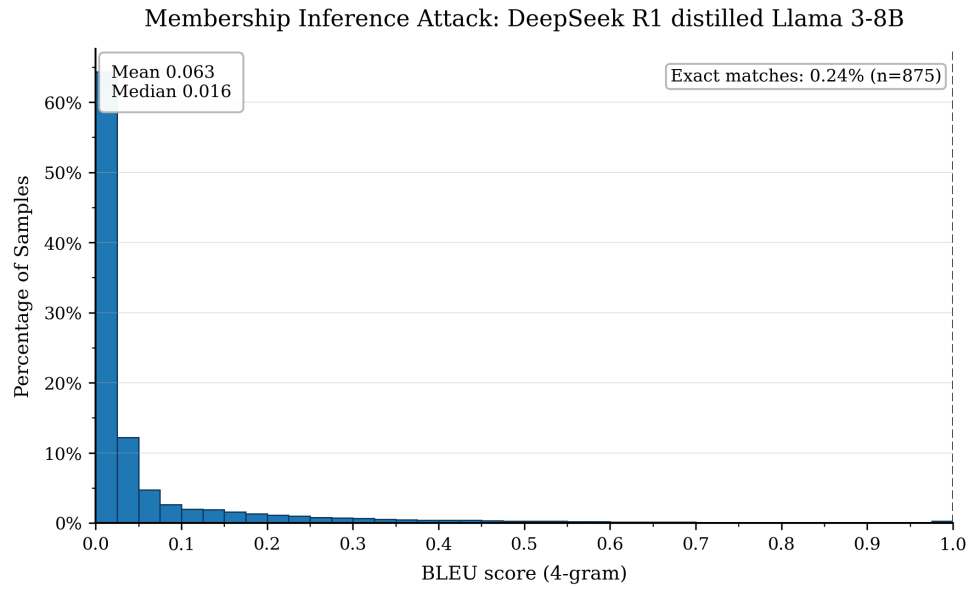


Figure 4.10: Membership Inference Attack Result for R1-distilled Llama3-8B

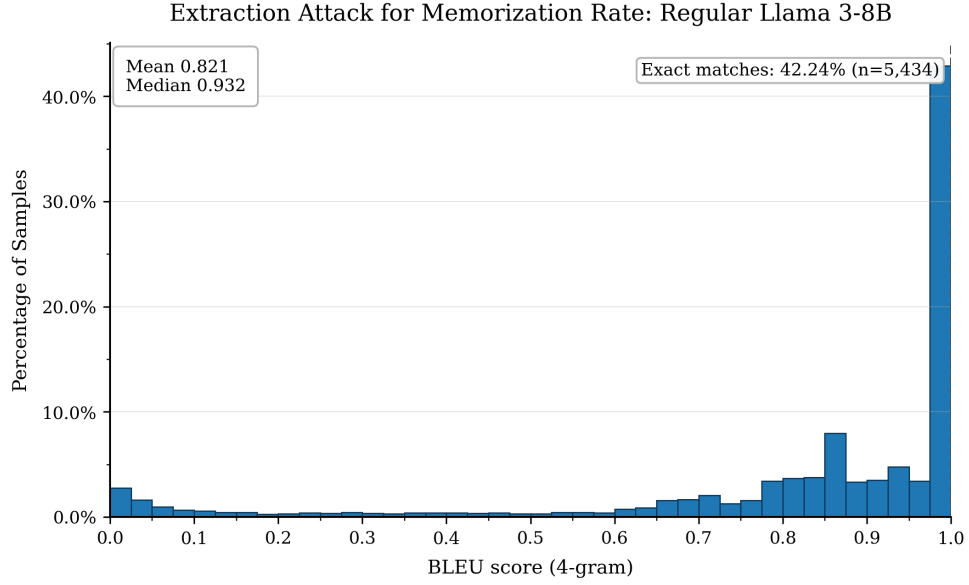


Figure 4.11: Extraction Attack for Memorization Rate LLama3-8B

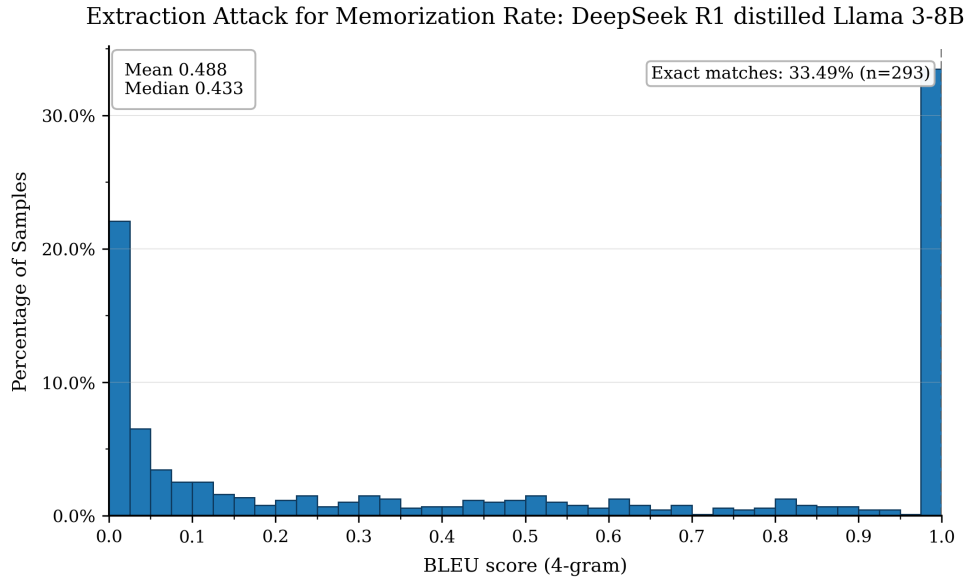


Figure 4.12: Extraction Attack for Memorization Rate R1-distilled LLama3-8B

against memorization that are resource-intensive or causing performance degradations. In contrast, distillation is a straightforward technique that often yields a faster model, while reduces the model’s verbatim recall. However, our investigation is preliminary, and the exact mechanism by which distillation tackles memorization is not yet clear, and further analysis is still required to validate the hypothesis. There are different factors that can have influences on memorization, for example, the choice of the teacher model, the capacity and size of the student model, and the distillation hyper-parameters might all affect the outcome.

Distillation might be effective in reducing memorization, but it does not eliminate memorization completely, as there is still a 34% extraction rate for the training set, meaning that the distilled model can still leak training snippets. Therefore, distillation should be considered as a component in a more comprehensive mitigation approach to further reduce the memorization. Overall, our findings suggest that distillation is a promising direction for reducing memorization in LLMs, while balancing cost and effectiveness. Further research can focus on refining this approach and better understanding the mechanism, to investigate how to maximize the ”unlearning” benefits without compromising model capability.

Research Question Summary 3

Knowledge distillation can be utilized as a cost-effective mitigation method.

4.4 Implications

The results of this work have important implications for both researchers and industrial practitioners.

Our findings confirms that unintended memorization remains a persistent issue even in the latest generation of models. Moreover, our analysis suggests that memorization is not only a function of a model’s parameter size, but also depends on the model’s architecture and designed applications. For example, we observed that StarCoder2 leaked more code examples than a similar sized Llama3 model, indicating that domain-specific training can amplify memorization on domain-specific knowledge. This result suggested practitioners investigate memorization across different models and different field of studies, rather than relying on parameter size alone.

Our results also offer a more granular view of what gets memorized, which has implications for how we evaluate and mitigate memorization towards different types of data. There

are certain types of code content that are more prone to memorization, particularly highly repetitive, boilerplate data, such as license text and documentation strings are frequently extracted by models, whereas more complex code logic and test code are memorized at a much lower rate. For researchers, this suggests that evaluating memorization should incorporate the target data characteristics. Future benchmarks could explicitly include different code categories or complexity levels to measure memorization in a finer-granularity. Our work also highlighted the importance of developing quantitative measures.

From an industry perspective, code-focused LLMs pose practical risks. Practitioners should impose stricter review and auditing of model outputs before use, and providers should curate training data to exclude sensitive content where possible. Repositories can embed secrets (API keys, credentials, personal data) in code or comments, and a memorizing model can unintentionally expose these. The memorization-revealing pipeline can be adopted as part of validation process as well.

Our findings also point toward possible mitigation strategies that guarantees performance without excessive cost. The distilled model has a lower extraction rate, which highlights the potential of knowledge distillation as an effective component in a more comprehensive mitigation approach.

Chapter 5

Threats to Validity

5.1 Internal Threats

One concern is the unfixing variables in our experiments. For example, while we compared models of different sizes, these models also differ in architecture and training data, and it is possible that the observed memorization differences are partly due to these factors. We constructed a novel independent dataset, but we cannot be certain that no overlap nor advantage exists in some models. Additionally, the attack pipeline is intrinsically arbitrary, and may not reveal all memorized content consistently.

5.2 External Threats

Another concern is with the scope of our study. We evaluated up to 8 billion parameters, which is comparably small given that some industrial models have hundreds of billions of parameters. It remains a question whether our observations would remain valid for bigger models. Additionally, we did not include proprietary LLMs such as the GPT-family or Claude models. Moreover, our mitigation experiment using knowledge distillation was conducted on a single teacher-student model pair, meaning the effectiveness of distillation in reducing memorization needs to be verified across more models and settings.

5.3 Construct Threats

The measurement of memorization contains certain ambiguity, as there is no universal definition of memorization in the context of language models, so we conducted it through the attack pipeline that we adopted. However, the threshold for memorization revealing can still be adjusted and optimized. The robustness can be potentially improved with a optimized set of parameters. Our use of a 4-gram BLEU can pose certain limitations, a high BLEU score between generated code and a known training snippet can suggest verbatim memorization, but memorization can take more subtle forms, that a BLEU score might not fully capture.

Chapter 6

Conclusion

6.1 Conclusion

This thesis was designed to explore the prevalence and characteristics of training data memorization in LLMs. We adopted and extended an experimental pipeline inspired by prior work, and conducted experiments on a carefully curated, independent Python code dataset.

Through membership inference attack and data extraction attack, we revealed memorization in state-of-the-art models, indicating that memorization is a persistent issue, and the revealing pipeline remains valid. We observed a content dependent pattern, suggesting licenses is the most memorized category, and testing is the least memorized, and in near realistic scenarios, code memorization could be triggered. We also found that knowledge distillation can meaningfully reduce memorization, as the distilled Llama3-8B revealed fewer memorized examples, without severe performance loss.

Overall, this study provides an empirical assessment for existing work, and establishes a baseline for evaluating, understanding, and mitigating memorization in LLMs.

6.2 Future Work

Building on this thesis, there are several promising directions for future research to explore and deepen our understanding of memorization in LLMs.

Researchers can extend the evaluation to larger-scale models, our findings suggests there was no dramatic changes between StarCoder2-3B and 7B models, but it remains unclear if the same finding holds for bigger models. A code model that is finetuned on the Llama3 family (e.g., LlamaCoder) can also be compared side-by-side with the StarCoder2 model.

Another direction for future work, is to improve the dataset design and evaluation technique. In our study, we curated a novel dataset to reduce bias, but future researchers could design a more comprehensive benchmarks for memorization revealing purposes, for example, including programming languages other than python. The memorization pipeline can also be optimized and tuned to improve the efficiency, for instance, effects from various parameters including snippet length, and evaluating metrics can still be further investigated.

Future work can also focus on refining mitigation techniques towards memorization. our preliminary findings demonstrated promising results that knowledge distillation can reduce the rate of memorized outputs. However, more research is preferred on the effect of knowledge distillation, for instance, understanding which aspects of the distillation process contribute the most to the mitigation of memorization. Besides, it would be worthy to explore if knowledge distillation can be combined with other mitigation method to form a mitigation pipeline. These directions are all promising extensions of our work.

References

- [1] L. Gao *et al.*, *The Pile: An 800GB Dataset of Diverse Text for Language Modeling*, Dec. 2020. DOI: [10.48550/arXiv.2101.00027](https://doi.org/10.48550/arXiv.2101.00027).
- [2] Y. Li *et al.*, “Competition-Level Code Generation with AlphaCode,” *Science*, vol. 378, no. 6624, pp. 1092–1097, Dec. 2022. DOI: [10.1126/science.abq1158](https://doi.org/10.1126/science.abq1158).
- [3] A. Al-Kaswan, M. Izadi, and A. van Deursen, “Traces of Memorisation in Large Language Models for Code,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE ’24, New York, NY, USA: Association for Computing Machinery, Apr. 2024, pp. 1–12. DOI: [10.1145/3597503.3639133](https://doi.org/10.1145/3597503.3639133).
- [4] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, “On the naturalness of software,” *Commun. ACM*, vol. 59, no. 5, pp. 122–131, Apr. 2016. DOI: [10.1145/2902362](https://doi.org/10.1145/2902362).
- [5] N. Carlini, C. Liu, Ú. Erlingsson, J. Kos, and D. Song, *The Secret Sharer: Evaluating and Testing Unintended Memorization in Neural Networks*, Jul. 2019. DOI: [10.48550/arXiv.1802.08232](https://doi.org/10.48550/arXiv.1802.08232).
- [6] N. Carlini *et al.*, *Extracting Training Data from Large Language Models*, Jun. 2021. DOI: [10.48550/arXiv.2012.07805](https://doi.org/10.48550/arXiv.2012.07805).
- [7] Z. Yang, Z. Zhao, C. Wang, J. Shi, D. Kim, D. Han, and D. Lo, “Unveiling Memorization in Code Models,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, Apr. 2024, pp. 1–13. DOI: [10.1145/3597503.3639074](https://doi.org/10.1145/3597503.3639074).
- [8] V. Smith, A. S. Shamsabadi, C. Ashurst, and A. Weller, *Identifying and Mitigating Privacy Risks Stemming from Language Models: A Survey*, Jun. 2024. DOI: [10.48550/arXiv.2310.01424](https://doi.org/10.48550/arXiv.2310.01424).
- [9] M. Nasr, N. Carlini, J. Hayase, M. Jagielski, A. F. Cooper, D. Ippolito, C. A. Choquette-Choo, E. Wallace, F. Tramèr, and K. Lee, *Scalable Extraction of Training Data from (Production) Language Models*, Nov. 2023. DOI: [10.48550/arXiv.2311.17035](https://doi.org/10.48550/arXiv.2311.17035).

- [10] D. Ippolito, F. Tramèr, M. Nasr, C. Zhang, M. Jagielski, K. Lee, C. A. Choquette-Choo, and N. Carlini, *Preventing Verbatim Memorization in Language Models Gives a False Sense of Privacy*, Sep. 2023. DOI: [10.48550/arXiv.2210.17546](https://doi.org/10.48550/arXiv.2210.17546).
- [11] W. Xu, K. Gao, H. He, and M. Zhou, *LiCoEval: Evaluating LLMs on License Compliance in Code Generation*, Feb. 2025. DOI: [10.48550/arXiv.2408.02487](https://doi.org/10.48550/arXiv.2408.02487).
- [12] F. Salerno, A. Al-Kaswan, and M. Izadi, *How Much Do Code Language Models Remember? An Investigation on Data Extraction Attacks before and after Fine-tuning*, Feb. 2025. DOI: [10.48550/arXiv.2501.17501](https://doi.org/10.48550/arXiv.2501.17501).
- [13] V. Feldman, *Does Learning Require Memorization? A Short Tale about a Long Tail*, Jan. 2021. DOI: [10.48550/arXiv.1906.05271](https://doi.org/10.48550/arXiv.1906.05271).
- [14] Y. Guo, M. Guo, J. Su, Z. Yang, M. Zhu, H. Li, M. Qiu, and S. S. Liu, *Bias in Large Language Models: Origin, Evaluation, and Mitigation*, Nov. 2024. DOI: [10.48550/arXiv.2411.10915](https://doi.org/10.48550/arXiv.2411.10915).
- [15] A. Satvaty, S. Verberne, and F. Turkmen, *Undesirable Memorization in Large Language Models: A Survey*, Mar. 2025. DOI: [10.48550/arXiv.2410.02650](https://doi.org/10.48550/arXiv.2410.02650).
- [16] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, “Membership Inference Attacks Against Machine Learning Models,” in *2017 IEEE Symposium on Security and Privacy (SP)*, May 2017, pp. 3–18. DOI: [10.1109/SP.2017.41](https://doi.org/10.1109/SP.2017.41).
- [17] Y. Chen, J. Chen, Y. Weng, C. Chang, D. Yu, and G. Lin, *AdaMixup: A Dynamic Defense Framework for Membership Inference Attack Mitigation*, Jan. 2025. DOI: [10.48550/arXiv.2501.02182](https://doi.org/10.48550/arXiv.2501.02182).
- [18] N. Carlini, S. Chien, M. Nasr, S. Song, A. Terzis, and F. Tramèr, “Membership Inference Attacks From First Principles,” in *2022 IEEE Symposium on Security and Privacy (SP)*, May 2022, pp. 1897–1914. DOI: [10.1109/SP46214.2022.9833649](https://doi.org/10.1109/SP46214.2022.9833649).
- [19] M. Duan, A. Suri, N. Mireshghallah, S. Min, W. Shi, L. Zettlemoyer, Y. Tsvetkov, Y. Choi, D. Evans, and H. Hajishirzi, *Do Membership Inference Attacks Work on Large Language Models?* Sep. 2024. DOI: [10.48550/arXiv.2402.07841](https://doi.org/10.48550/arXiv.2402.07841).
- [20] A. Al-Kaswan, M. Izadi, and A. v. Deursen, *Targeted Attack on GPT-Neo for the SATML Language Model Data Extraction Challenge*, Feb. 2023. DOI: [10.48550/arXiv.2302.07735](https://doi.org/10.48550/arXiv.2302.07735).
- [21] R. Rabin, S. McGregor, and N. Judd, *Malicious and Unintentional Disclosure Risks in Large Language Models for Code Generation*, Mar. 2025. DOI: [10.48550/arXiv.2503.22760](https://doi.org/10.48550/arXiv.2503.22760).

- [22] K. Lee, D. Ippolito, A. Nystrom, C. Zhang, D. Eck, C. Callison-Burch, and N. Carlini, *Deduplicating Training Data Makes Language Models Better*, Mar. 2022. DOI: [10.48550/arXiv.2107.06499](https://doi.org/10.48550/arXiv.2107.06499).
- [23] Y. Shanmugarasa, M. Ding, M. A. P. Chamikara, and T. Rakotoarivelo, *SoK: The Privacy Paradox of Large Language Models: Advancements, Privacy Risks, and Mitigation*, Jun. 2025. DOI: [10.1145/3708821.3733888](https://doi.org/10.1145/3708821.3733888).
- [24] A. Hans *et al.*, *Be like a Goldfish, Don't Memorize! Mitigating Memorization in Generative LLMs*, Nov. 2024. DOI: [10.48550/arXiv.2406.10209](https://doi.org/10.48550/arXiv.2406.10209).
- [25] M. Sakarvadia, A. Ajith, A. Khan, N. Hudson, C. Geniesse, K. Chard, Y. Yang, I. Foster, and M. W. Mahoney, *Mitigating Memorization In Language Models*, Jan. 2025. DOI: [10.48550/arXiv.2410.02159](https://doi.org/10.48550/arXiv.2410.02159).
- [26] G. Hinton, O. Vinyals, and J. Dean, *Distilling the Knowledge in a Neural Network*, Mar. 2015. DOI: [10.48550/arXiv.1503.02531](https://doi.org/10.48550/arXiv.1503.02531).
- [27] X. Xu, M. Li, C. Tao, T. Shen, R. Cheng, J. Li, C. Xu, D. Tao, and T. Zhou, *A Survey on Knowledge Distillation of Large Language Models*, Oct. 2024. DOI: [10.48550/arXiv.2402.13116](https://doi.org/10.48550/arXiv.2402.13116).
- [28] Y. R. Dong, H. Lin, M. Belkin, R. Huerta, and I. Vulić, *Unmemorization in Large Language Models via Self-Distillation and Deliberate Imagination*, Feb. 2024. DOI: [10.48550/arXiv.2402.10052](https://doi.org/10.48550/arXiv.2402.10052).
- [29] A. Lozhkov *et al.*, *StarCoder 2 and The Stack v2: The Next Generation*, Feb. 2024. DOI: [10.48550/arXiv.2402.19173](https://doi.org/10.48550/arXiv.2402.19173).
- [30] A. Grattafiori *et al.*, *The Llama 3 Herd of Models*, Nov. 2024. DOI: [10.48550/arXiv.2407.21783](https://doi.org/10.48550/arXiv.2407.21783).
- [31] DeepSeek-AI *et al.*, *DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning*, Jan. 2025. DOI: [10.48550/arXiv.2501.12948](https://doi.org/10.48550/arXiv.2501.12948).
- [32] S. Fernandes and J. Bernardino, “What is BigQuery?” In *Proceedings of the 19th International Database Engineering & Applications Symposium*, ser. IDEAS '15, New York, NY, USA: Association for Computing Machinery, Jul. 2015, pp. 202–203. DOI: [10.1145/2790755.2790797](https://doi.org/10.1145/2790755.2790797).