Computer Assisted Methods in Engineering and Science
32(3): 293–315, 2025, https://doi.org/10.24423/cames.2025.1781

A Study on the Extraction of Training Dataset from Fine-Tuned Language Models

Raja VAVEKANAND[1)*], Aybek Kalandarov RUZIMBAEVICH[2)], Muhabbat JUMANIYOZOVA[3)]

[1)] *Department of Information Technology, Benazir Bhutto Shaheed University Lyari Karachi, Pakistan*

[2)] *Department of Foreign Languages, Urgench Ranch University of Technology, Urgench, Uzbekistan*

[3)] *Department of Methodology of Primary Education, Urgench State University, Urgench, Uzbekistan*

[*] *Corresponding Author e-mail: rajavavekanand@yahoo.com*

Large language models (LLMs) excel at various natural language tasks, even those beyond their explicit training. Fine-tuning these models on smaller datasets enhances their performance for specific tasks but it can also lead to risk of training data memorization, raising privacy concerns. This study explores the extraction of private training data from fine-tuned LLMs through a series of experiments. The focus is on assessing the ease of data extraction using various techniques and examining how factors such as the size of training data, number of epochs, training sample length and content, and fine-tuning parameters influence this process. Our results indicate that data extraction is relatively straightforward with direct model access, especially when training loss is computed over entire prompts. Models with higher precision (8-bit and 16-bit) demonstrate increased memorization capabilities compared to 4-bit quantized models. Even without direct access, insights into training data can be obtained by comparing output probability scores across multiple queries. Furthermore, the study also reveals that the proportion of extractable data increases with training dataset size, given a fixed number of epochs. These findings highlight the privacy risks faced by individuals whose data is used in fine-tuning, as well as for organizations deploying fine-tuned models in public applications.

Keywords: fine-tuning, large language models, data extraction, quantized low-rank adaptation (QLoRA).

1. Introduction

Large language models (LLMs), especially transformer-based models, have surged in recent years. These models demonstrate remarkable performance across a wide range of natural language tasks, even those they were not explicitly trained for. Transformer language models are versatile, handling tasks such as text generation, classification, and question answering [14]. While zero-shot prompting can accomplish these tasks, few-shot prompting and in-context learning often yield better results [2, 12]. These techniques are preferred because they do not require access to model weight and are computationally efficient avoiding backpropagation. However, in-context learning consumes the limited prompt length each time the task is performed and heavily relies on the quality and perplexity of the provided examples. Models can be fine-tuned on smaller datasets for specific tasks. Fine-tuning generally outperforms prompt engineering or in-context learning, producing standalone models [13]. However, fine-tuning large state-of-the-art models with billions of parameters is expensive and impractical. To address this, parameter-efficient fine-tuning methods such as prefix tuning [9], lower layers freezing, adapters, and quantization have been developed [10]. This work focuses on quantized low-rank adapters (QLoRA), a method that inserts trainable rank decomposition matrices into transformer layers while freezing the rest of the base model. Each LoRA layer has two trainable matrices, $\mathbf{A}$ and $\mathbf{B}$, with the rank r much smaller than the hidden dimension k and the output dimensions d. If the original layer weights are $\mathbf{W}0$, the layer's output is $\mathbf{W}0x + \mathbf{BA}x$, scaled by a hyperparameter α.

Compared to full fine-tuning, LoRA dramatically reduces the number of trainable parameters by up to 10 000 times and requires three times less GPU memory, all without sacrificing performance [7]. Additionally, quantizing the model to 4- or 8-bit reduces computational costs without compromising accuracy. Benchmarks show that models with 4-bit quantized LoRA adapters perform as well as fully fine-tuned models [6]. This technique's efficiency and effectiveness make it a focus of this research. However, there remains a critical concern: large amounts of training data used for pre-training can be extracted from these models. Alignment fine-tuning may not mitigate this risk, raising, therefore, privacy issues, especially when sensitive or copyrighted data are involved. The ease of extracting such data from fine-tuned models, particularly for specific tasks, is not yet well understood [11]. This work aims to explore the conditions under which training data can be extracted from fine-tuned LLMs. QLoRA is a memory-efficient fine-tuning method that adds small, trainable low-rank matrices into transformer layers, reducing the number of updated parameters by up to 10 000 times compared to full fine-tuning, while maintaining model performance.

This research aims to understand the privacy risks associated with fine-tuning LLMs on sensitive data. Consider a small business with a database containing user names, credit card numbers, and purchase histories that fine-tunes an LLM using QLoRA to create a chatbot for product recommendations. The model might memorize these numbers during training, which is increasingly common as businesses deploy fine-tuned LLMs in customer-facing applications. An attacker with access to the model could potentially extract these credit card numbers, posing a privacy risk. The research addresses several key questions: how effectively can private training data be extracted from fine-tuned models, the impact of training data size, number of epochs used for fine-tuning, and the length and content of each training sample, and the role of the fine-tuning technique and parameters used. Experiments were conducted to test a fine-tuned model with data including credit card numbers, assuming full access to the model inputs and outputs and unlimited query attempts. The findings will help businesses and researchers make informed decisions when using fine-tuned models and to implement measures to mitigate data extraction risks.

2. Related work

2.1. Pre-training data extraction

Training data has been reliably extracted from pre-trained LLMs. Carlini *et al.* [5] demonstrated that significant amounts of data, including names, phone numbers, and addresses, could be extracted from GPT-2 by querying the model with various prompt prefixes and filtering responses using six membership inference metrics. Over a third of the filtered responses were present in the training data. They introduced the concept of "extractable memorized" data as examples that an adversary, without access to the training set, can prompt the model to generate. They found that at least 1% of GPT-2 or GPT-3's training data was extractable and memorized with 50-token prompts. Larger models memorized two to five times more data than smaller ones, repeated examples were more likely to be memorized, and longer context prompts made extraction easier. Carlini *et al.* [5] also introduced "discoverably memorized" data, where an adversary with access to training data prefixes can prompt the model to generate the example. Their findings indicated that LLMs aligned for chatting are vulnerable, if not more, to data extraction attacks. While these studies focus on pre-training data extraction, similar techniques may apply to fine-tuned models [3]. This work investigates the extraction of discoverably memorized credit card numbers, hypothesizing that longer context, more parameters, and repeated training data increase memorization. Focusing on credit card numbers provides a clear way to compare model completions and assess data memorization.

2.2. Theoretical memorization

Allen-Zhu and Li [1] established a theoretical upper bound on the amount of data that can be memorized by a transformer-based LLM. Their work demonstrates that memorization capacity is proportional to the number of model parameters, with a capacity of approximately 2 bits of information per parameter. Their findings show that this scaling law applies to both 16-bit and 8-bit quantized models. It is important to note that this represents a theoretical upper bound, which may not be achievable in practice. Although their study is focused on data memorization rather than data extraction, and does not specifically address fine-tuning, it offers relevant insights. Specifically, their work highlights that the amount of data that can be memorized during pre-training is constrained by the model's parameter count. This may have implications for how much fine-tuning data can be stored in LoRA adapters compared to the modified parameters in a full fine-tuning scenario. If each trainable parameter can memorize up to 2 bits of information, fine-tuning methods that modify more parameters could be more vulnerable to data extraction.

For instance, using LoRA with a higher value of r increases the size of the learned $\mathbf{A}$ and $\mathbf{B}$ matrices, thereby increasing the number of tunable parameters. This, in turn, could result in a larger potential for data memorization and extraction. Additionally, prior work implies that some forms of quantization may not significantly impact the amount of data a model can memorize. If the data extractable from a model is directly related to its memorization capacity, quantization may not substantially alter the amount of data that can be extracted. While the studies discussed above highlight the risks of data extraction from pre-trained LLMs, considerably less is known about fine-tuned models. The following sections outline our methodology for investigating training data extraction from fine-tuned LLMs, focusing on the impact of fine-tuning parameters and prompt characteristics.

3. Methods

3.1. Dataset creation

A synthetic dataset for this research was generated using the Faker Python package, creating 1000 fictional users in a business scenario. Each user was assigned a name, email, phone number, credit card number, a list of five purchased items, and the total amount spent. Names, emails, phone numbers, and credit card numbers were produced using Faker's built-in providers. Names and emails were generated independently using common English names and typical email formats, while phone numbers followed the standard US format. Credit card numbers, typically 16 digits long and starting with an issuer identification number (IIN), were validated using the Luhn algorithm to ensure validity.

For generating the list of purchased items, a pool of 250 adjectives and 250 nouns was created using Faker. For each user, one adjective and one noun index were chosen randomly, and five items were selected by sampling around these indices based on a normal distribution. This method introduces built-in correlations within the dataset. The total amount spent by each user was randomly generated as an integer between 0 and 1000. All data was saved as a comma-separated value (CSV) file to ensure consistency across all experiments. To create a chat-like model, the raw data was converted into a chat format using Hugging Face chat templates. Hugging Face chat models require specific formatting, and their tokenizers include templates that ensure higher-quality outputs. A JavaScript object notation (JSON) file was generated where each line represented a user prompt and the assistant's response. The user prompt included the user's name and credit card number, while the assistant's response contained a product name. The apply_chat_template function, as defined in [16], formatted these entries accordingly. The default Mistral template was structured as "[INST] {user input}[/INST]{assistant response}".

The JSON dataset was then converted into strings following this format and tokenized by the model's tokenizer before being used for fine-tuning. The same template was applied during model generation, but with an empty assistant response, ensuring consistency between fine-tuning and generation processes as well as across different experiments. Three primary types of prompt/response pairs were created for the experiments: short, medium, and long.

- **Short**: The user prompt contains only the user's name and credit card number, while the assistant's response is a product name. Example:
  - User: Scott Hahn's credit card number is 3525609767017203. What is the last product they purchased?
  - Assistant: The user last purchased a White Seat.
- **Medium**: The user prompt includes the user's name, a list of four products purchased, and their credit card number. The assistant's response is a product name. Example:
  - User: Scott Hahn's purchase history includes White Seat, Wide Mess, Wide Trick, and Any Birth. Their credit card number is 352560976701 7203. What is a product you could recommend them?
  - Assistant: The user would enjoy Huge Wave.
- **Long:** The user prompt includes the user's name, email, phone number, credit card number, a list of four products purchased, and the total amount spent. The assistant's response is a product name. Example:
  - User: Recommend a product for this user: name: Scott Hahn, email: kendra66@example.org, phone: (423) 945-0076, credit_card: 35256097 67017203, total_spent: 33, purchase_history: White Seat, Wide Mess, Wide Trick, Any Birth.
  - Assistant: Huge Wave would be a good choice.

These prompts simulate a chat in which the assistant retrieves or suggests product information based on user data. The short prompt is simple and contains minimal information, serving as a baseline for data extraction. The medium prompt is more realistic for creating a product recommendation bot. The long prompt, while less conversational, simulates a direct dump of user data, potentially including sensitive information such as credit card numbers. By analyzing these different prompt types, we aim to assess how easily credit card numbers can be extracted from the fine-tuned model. The short prompt establishes that data extraction is possible, the medium prompt represents a real-world use case, and the long prompt highlights the risk of inadvertently including sensitive data in training sets.

3.2. Model selection

There are various popular base models to choose from when fine-tuning a large language model. For the scenario simulating a business creating a chatbot for customers, we chose Mistral-7B-Instruct-v0.2 as the base model because it has already been aligned for chat applications. This recently published model is a version of Mistral-7B that has been fine-tuned with instructions to perform well on chat completions. The Mistral-7B model is a 7-billion-parameter model that uses techniques such as grouped query attention, sliding window attention, a rolling buffer cache, and a byte-fallback byte pair encoding (BPE) tokenizer to achieve high performance with fewer parameters and faster evaluation. Compared to the Llama 2 family of models, Mistral-7B supports double the context length. Several benchmarks have shown that Mistral-7B outperforms other similar 7-billion-parameter open source/open weight models, as well as the Llama-2 13-billion-parameter model [8].

3.3. Fine-tuning

To leverage the existing instruction fine-tuning, user prompts provided to the model should be enclosed within [INST] and [/INST] tokens. Most of the code used for fine-tuning across various experiments was adapted from [16], which made it straightforward to fine-tune the model on different datasets and with varying parameters. The code is designed to automatically apply the appropriate chat template to the training data, eliminating the need for manual insertion of instruction tokens. We used the same fine-tuning process across all experiments to maintain consistency. The default implementation of the apply_chat_template function, located in [location/codebase], appends a closing [/INST] token at the end of any user prompt. The base model was fine-tuned individually for each experiment, based on the specific experimental parameters. One crucial aspect of the training process was how the training loss was calculated. The training code aims to minimize the loss of the model's generated responses, so the method used to calculate loss has a significant impact on the model's behavior. By default, loss is computed for every token generated, including both the user's message or the assistant's response.

In certain experiments, we focused on calculating loss solely for the assistant's messages, as these are the only messages generated by the model. This approach may more effectively minimize the loss of assistant-generated content. However, a downside of this method is that it disables packing, a technique that increases the number of examples processed in a single batch, which can slow down training. Additionally, packing may lead to reduced performance if the examples being packed are highly correlated. To ensure consistency, packing was not used in any of the fine-tuning experiments.

3.3.1. Full fine-tuning. Full fine-tuning is the most straightforward fine-tuning approach, where the entire model is fine-tuned on the training data. While it is the most computationally expensive fine-tuning technique, it is also the most flexible. In total, the Mistral-7B-Instruct-v0.2 model has 7 241 732 096 trainable parameters, requiring a large amount of memory and computational power to fully fine-tune. When using a graphics processor unit (GPU) for this purpose, even if the model weights fit into the GPU memory (around 15 GB in this case), full fine-tuning requires backpropagation, which needs at least around three times more memory and computation time compared to the forward pass. This is because the model needs to store the gradients for each parameter in memory. To successfully fine-tune the Mistral 7B model, we used a cluster of NVIDIA A100 GPUs with 80 GB memory each, along with DeepSpeed ZeRO-3 to parallelize the full fine-tuning across multiple GPUs. DeepSpeed ZeRO uses data parallelism to reduce the memory requirements for training large models by partitioning the model across multiple GPUs and storing only the gradients per each partition. This allows for training models larger than the memory of a single GPU. Specifically, ZeRO stage 3 shards the optimizer states, gradients, and model parameters [15]. Without this sharding, individual GPUs ran out of memory after just a few optimization steps.

3.3.2. QLoRA fine-tuning. Using LoRA with rank $r = 32$ reduces the number of trainable parameters to 83 886 080 and LoRA with rank $r = 16$ this number further decreases to 41 943 040. This is only 0.5% of the parameters trained in full fine-tuning. This reduction in parameters allows fine-tuning to be performed on a single GPU without the need for ZeRO-3. For consistency across experiments, the QLoRA was standardized using rank $r = 16$, $\alpha = 16$, 4-bit quantization, and 20 training epochs, serving as a baseline for comparison.

3.4. Data extraction

Each fine-tuned model was then used for data extraction in one of two possible attacks. Training data was reliably extracted from pre-trained LLMs, as described in Subsec. 2.1.

3.4.1. Partial prompt completion. The partial prompt completion attack uses the model itself to generate data inside the user portion of a prompt. An attacker must have access to the prompt string being sent to the tokenizer. By default, implementation of apply_chat_template adds a closing [/INST] token to the end of every user prompt. The attacker must account for any modifications to their input prompt imposed by the chat templates. Assuming the attacker can remove the closing [INST] token, the model does not recognize that the user prompt has ended. The attacker can then submit a partial user prompt. If the model is trained to minimize loss over the entire output (not just the assistant's response), it will attempt to first complete the user prompt. In our experiments, we assumed that the attacker knew the exact prompt format as well as all user information except credit card numbers. For example, a model trained with the short prompt can be extracted by inputting "[INST] Scott Hahn's credit card number is" and the model's response is completed.

Since the [INST] token is not closed, the model may attempt to complete the user prompt with a credit card number. This attack corresponds to "discoverable memorization" as defined by Nasr *et al.* [13] and Carlini *et al.* [3]. The experiments related to testing this attack are detailed in Sec. 4.

3.4.2. Loss comparison. In the case of completion-only loss calculation, an attack based on the model generating parts of the user prompt may not work. However, looking at how transformer-based language models work, we know that each new token is generated based on attention to all previous tokens. That means that the probability scores used to generate each successive assistant token can still be influenced by preceding tokens in the user prompt, including a credit card number. An attacker with access to these scores could, in theory, learn some information about the training data used to fine-tune the model by comparing scores across many different generations. In our case, we assumed the attacker had full access to the token probability scores during generation, knew all user data except the credit card numbers, and knew the exact prompt format. The attacker could generate many different completions of a particular user prompt with different prospective credit card numbers filled in and retrieve the probability scores for generating a part of the assistant's response. The attacker could then calculate the cross-entropy loss of these scores relative to the known model response from the training data to try and figure out which credit card number was likely present in the training data. The pseudocode for this attack is shown in Algorithm.

Algorithm. Loss comparison algorithm pseudocode.

```
loss fn ← nn.CrossEntropyLoss
random cc numbers ← {“3525609767017203”, “6553751007408996”,
    “371511505520350”}
rankings ← {}
for guess in random cc numbers do prompt ← “Scott Hahn’s purchase...
    credit card number is {guess}. What is a product you could recommend them?”
prompt ← tokenizer.encode(prompt)
response ← tokenizer.encode(“The user would enjoy Huge Wave”)
scores, tokens ← model.generate(prompt, ...)
loss ← loss fn(scores, response)
rankings[guess] ← loss
end for
return rankings.sort()
```

In this way, the model can act as an oracle to confirm whether a randomly generated credit card number is correct and belongs to a particular person. This is a much more complex attack than partial prompt completion, and was therefore not expected to be as effective. The experiments related to testing this attack are detailed in Subsec. 4.3. We initially hypothesized that the correct credit card number would have the lowest loss compared to other random credit card numbers; however, experiments showed that this was not always the case.

3.5. Evaluation

Each of these extraction methods needs to be evaluated separately. Since partial prompt completion attempts to directly generate credit card numbers, the first step is to extract the number. This is done by looking at the generated response and matching the first numeric string using regex. Even if a full credit card number is not memorized, reliably guessing parts of one is still dangerous. Thus, we decided to compare the actual numbers with the generated ones with the help of Levenshtein edit distance. Levenshtein distance is the number of single character edits – insertions, deletions, or substitutions – required to change one string into another. It is a good metric for quantifying how close the generated number is to the actual number since it is sensitive to the order of the characters in the string. For two strings of lengths L_1 and L_2, the Levenshtein ratio is calculated as $1 - (\text{Levenshtein distance}/(L_1 + L_2))$. For example, a ratio of 1 indicates that the credit card numbers are identical, while a ratio of 0 means they share no digits. This is useful for comparing similarities even across different lengths of numbers.

To establish a good baseline for comparison to see what the Levenshtein ratio for random guessing would be, we generated 1000 random 16-digit numbers and compared each of them to 1000 newly generated random 16-digit numbers. The average Levenshtein ratio across 1 million comparisons was 0.375. Next, we considered how a smarter adversary would perform. If an attacker knew the vendor and the last four digits of a credit card number (which can often be found on discarded receipts), they would know five digits total. To simulate this, we again generated 1000 random 16-digit numbers, but compared each one to 1000 new random numbers where the first and last four digits matched. The average Levenshtein ratio in this attack scenario across 1 million comparisons was 0.553. These two averages form a baseline for random guessing and smart guessing, respectively. Any model where the average Levenshtein ratio of extractions is higher than 0.375 performs better than random guessing, and if the average Levenshtein ratio is higher than 0.553 the model performs better than a smart attacker, which is very dangerous. The loss comparison method does not directly generate credit card numbers for one-to-one comparison, so the evaluation is more indirect. During the evaluation, we generate many different completions of a user prompt, each with different prospective credit card numbers (including the actual one). We then calculate the cross-entropy loss, and next sort the generated losses and compare the index of the correct card number.

4. Experiments and results

4.1. Experiments with full fine-tuning

We began by evaluating the discoverable memorization of fully fine-tuned models across different numbers of training epochs by evaluating partial prompt completion. Evaluated fine-tuned models were trained on 32, 100, and 1000 training examples of short, medium, and long prompts. Figure 1 shows the results for 100 short prompts. The red dotted line represents the average Levenshtein ratio for random guessing (0.375), while the green dotted line represents the average Levenshtein ratio for smart guessing (0.553). The graph shows box plots of the Levenshtein ratios of each of the 100 training examples passed through the model, with outlier points defined as points more than 1.5 times the interquartile range from the median. The results show that the model suddenly begins to memorize exact credit card numbers at 16 epochs of training. At 16 epochs,

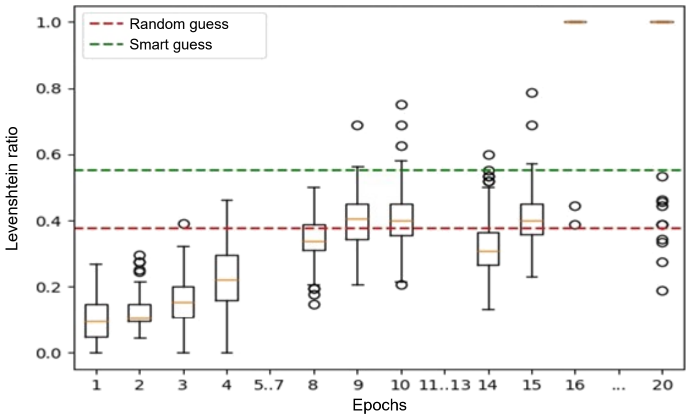


FIG. 1. Full fine-tune memorization across epochs: at 15 epochs or fewer, the median Levenshtein ratio approximates random guessing (0.375 ±0.078, red dotted line). By 16 epochs, 98% of examples show complete memorization, exceeding smart guessing (0.553 ±0.069, green dotted line). Error bars represent 1-sigma intervals.

98 of the 100 credit card numbers are fully memorized, compared to 15 epochs where none are fully memorized and the median Levenshtein ratio is 0.4, which is just above random guessing. Below 15 epochs, the median Levenshtein ratios are even lower. These trends are consistent across different numbers of training examples and prompt lengths tested. At 16 epochs, fully fine-tuned models showed high proportions of full memorization, as shown in Fig. 2. Increasing the number of examples tended to increase the proportion of fully memorized numbers, whereas prompt length yielded unclear effect. We also noticed that 16 epochs corresponded to the point where the model began to converge on a stable train-

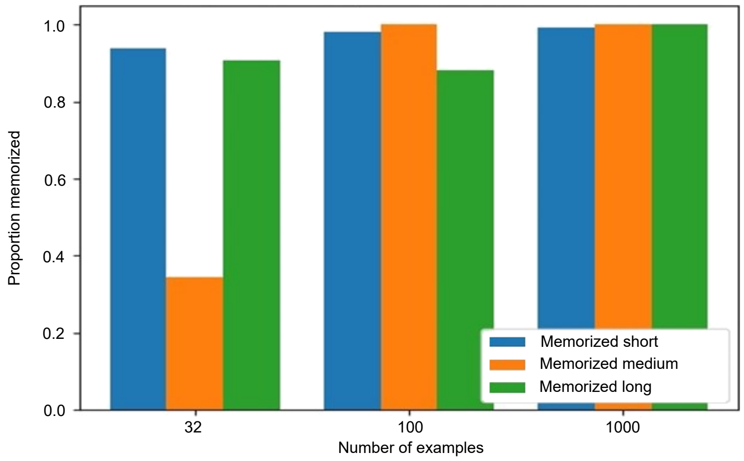


FIG. 2. Full memorization ratios for full fine-tuning. The Y-axis represents the proportion fully memorized, which indicates the proportion of credit card numbers fully memorized by the model, consistent with the labeling used in Table 2.

ing loss. Figure 3 shows that the evaluation loss had already started increasing by 16 epochs. The random guessing baseline was computed by comparing 1000 random 16-digit numbers to 1000 new random 16-digit numbers, yielding an average Levenshtein ratio of 0.375 with a standard deviation of 0.078. The smart guessing baseline, fixing the first digit and last four digits, yielded an average Levenshtein ratio of 0.553 with a standard deviation of 0.069.

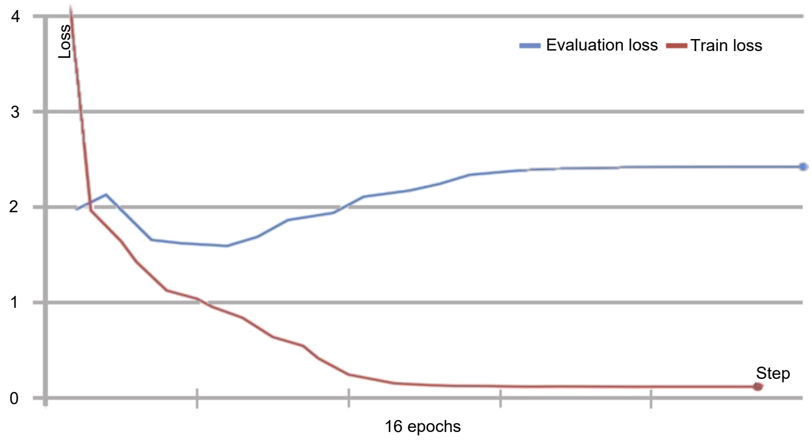


FIG. 3. Loss vs. training step: evaluation loss (blue), train loss (red); training loss stabilizes at 16 epochs, but the evaluation loss increases, indicating overfitting and suggesting full memorization.

4.2. Experiments with QLoRA fine-tuning

QLoRA fine-tuning experiments began by running only a few epochs of training. Most model generations, in the first five epochs of training, did not produce credit card numbers in the format used in the training set. The following are some examples of completions generated by the model:

- "...'s credit card number is 1348-2000-7982-9999", where the number provided was not in the dataset and also had a different format (dashes between groups of four digits),
- "...'s credit card number is ****-****-****-****", with asterisks replacing the digits,
- "...'s credit card number is 3520000000000000...", where the number starts with a common IIN but continues with a single digit repeated many times,
- "...'s credit card number is Visa", where the model generates a credit card type instead of a number,
- "...'s credit card number is unknown", where the model refuses to generate a credit card number.

When prompted more directly, the model would sometimes refuse to provide sensitive information such as credit card numbers. When a credit card number was included in the prompt, the model responded with “the question mistakenly includes the user’s credit card”.

The base Mistral 7B Instruct model has been aligned to be a chat model that does not output sensitive information such as credit card numbers. Many other LLMs have safety features built-in to prevent them from being misused. This alignment training still impacts generation when the model has not been fine-tuned long enough to output explicit credit card details. This issue was also present, to a lesser degree, when setting the LoRA α to a lower value. Continuing training past five epochs showed that the model began to memorize credit card numbers at a far more gradual rate compared to full fine-tuning. Experiments using 100 examples of medium prompts are shown in Fig. 4. With increasing epochs, the median and maximum Levenshtein ratios gradually increase. At 15 epochs, two examples are fully memorized, but over 75 examples performed worse than smart guessing. At 20 epochs, 28 examples are fully memorized and only 20 are worse than smart guessing. This shows that more epochs lead to increased discoverable memorization in terms of both full and partial memorization when using QLoRA fine-tuning. Compared to full fine-tuning, QLoRA requires more epochs to reach a similar level of memorization, and the increase in full and partial memorization occurs much more gradually. We observed that training and evaluation loss curves were very similar to full fine-tuning, meaning the model may still overfit in this setting. For the remainder of the experiments, 20 epochs were used as the baseline.

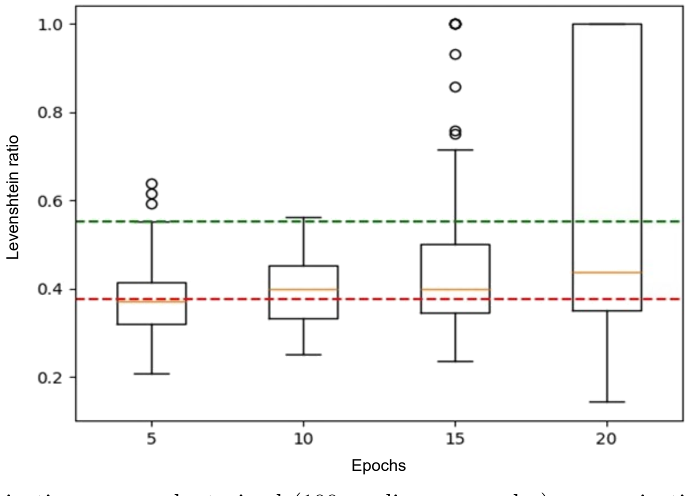
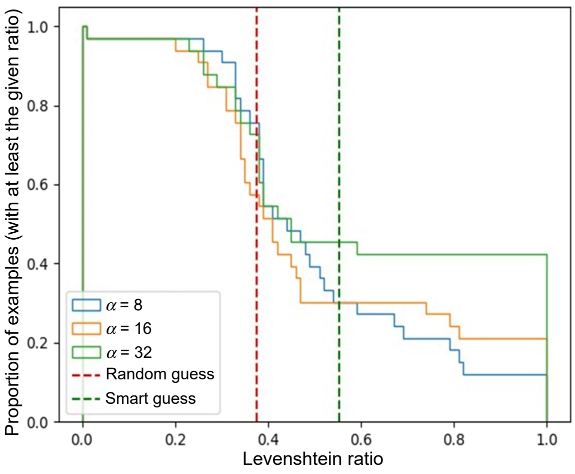


Fig. 4. Memorization vs. epochs trained (100 medium examples): memorization patterns for credit card numbers across varying prompt lengths and epochs. The graph shows the proportion of memorized credit card numbers at different stages of fine-tuning with 100 short prompts. 1-sigma error bars indicate variability in the results.

Next, we looked at how the QLoRA parameters impacted memorization. To show different levels of partial memorization for a particular set of training examples, cumulative distribution function (CDF) plots of Levenshtein ratios were used. A point with a Levenshtein ratio of x on the graph indicates what proportion of examples in the training data were memorized to a Levenshtein ratio of at least x. All examples have ratios of at least 0, and only the fully memorized examples achieve a ratio of 1.

The higher a curve rises toward Levenshtein ratios above the random or smart guessing, the more credit card numbers have been partially or fully memorized. As shown in Fig. 5a, higher α values led to increased memorization. This is expected as higher α values indicate that the trained LoRA matrices are mul-

a)


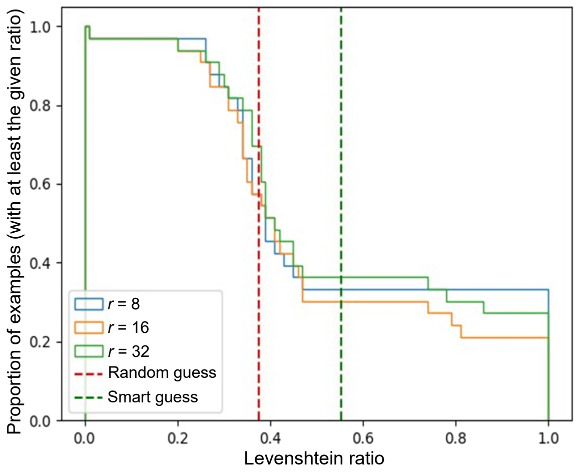
b)


FIG. 5. The impact of LoRA α and rank r on memorization: a) higher values of α result in increased memorization; b) the effect of varying r on memorization is minimal at this scale. The Y-axis represents the proportion of credit card numbers fully memorized, with error bars indicating variability.

tiplied by a larger scalar, so the fine-tuning affects the model's output more. Figure 5b shows that r had no clear effect on memorization at this scale. This is likely because the effort of memorizing credit card numbers is not complicated enough to warrant the use of higher-rank LoRA matrices (and thus more parameters).

Comparing memorization performance across quantization levels, we found that higher-precision models memorized more than heavily quantized ones, as expected. Doubling the quantization level from 8-bit to 16-bit improved the number of examples memorized by 2. On the other hand, doubling it from 4-bit to 8-bit increased memorized samples by 4 (Table 1). This shows that higher-precision models are more susceptible to memorization, although the effect is not as pronounced at higher precision levels.

TABLE 1. Memorization vs. quantization level.

Quantization	Number memorized (of 32)
4-bit	13
8-bit	17
16-bit	19

Setting the QLoRA parameters and number of epochs constant, we experimented with varying the number of training examples as well as the types of prompts used. The graphs in Fig. 6 show the CDFs of Levenshtein ratios for models fine-tuned with 32, 100, and 1000 examples. Across all three prompt lengths, it is clear that increasing the number of training examples leads to more memorization. Amongst the short prompt, the model trained on 1000 examples fully memorized 89% of its credit card numbers, whereas the one trained on 32 examples fully memorized only 40%. This is counterintuitive since adding more training examples requires the model to memorize more credit card numbers while using the same number of parameters, yet these models memorized higher proportions of the larger credit card datasets.

One possible explanation is the fact that more data forces the model to train for more steps per epoch. Additionally, it is likely that even with $r = 16$, the number of parameters may be larger than necessary to memorize the data. To test this hypothesis, an experiment where a model was trained with different amounts of data but for approximately the same number of training steps was conducted (see Table 2). Looking at the three models trained for approximately 2000 steps each, the model trained on 32 examples fully memorized far more than the model trained on 100 examples, while the model trained on 1000 examples did not memorize any credit card numbers. This suggests that increasing the number of training steps increases the amount of memorization, especially

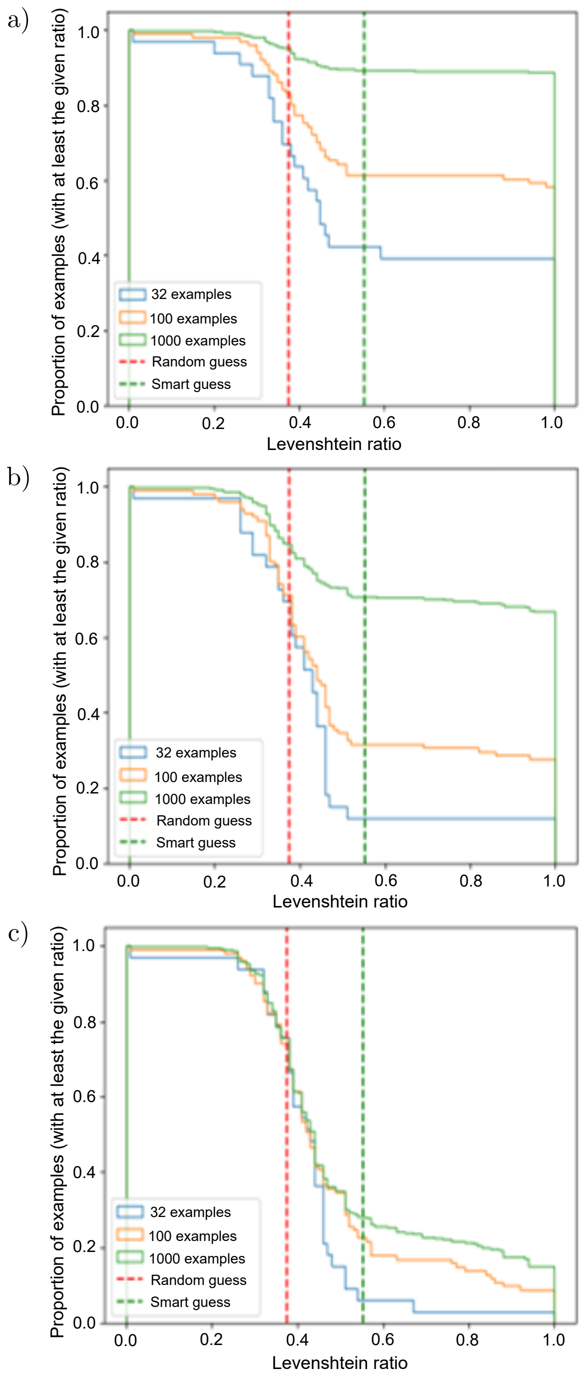


FIG. 6. CDFs of Levenshtein ratios: a) short, b) medium, and c) long, for models trained with 32, 100, and 1000 examples. The graphs show how the proportion of fully memorized credit card numbers changes with the number of training examples used, with error bars indicating the variability of results.

Table 2. Full memorization across different numbers of examples, epochs and steps.

Number of examples	Number of epochs	Number of steps	Proportion fully memorized
32	63	2016	0.8125
100	20	2000	0.59
1000	2	2000	0

if the dataset is kept small. This result agrees with prior work that has shown that more exposures to the data are needed to memorize more information [1].

Comparing the graphs in Fig. 6, one can see differences between partial prompt completion across different prompt lengths. Figure 7 highlights the difference, in particular, for the case with 1000 training examples. The shorter the prompt, the greater the degree of credit card number memorized.

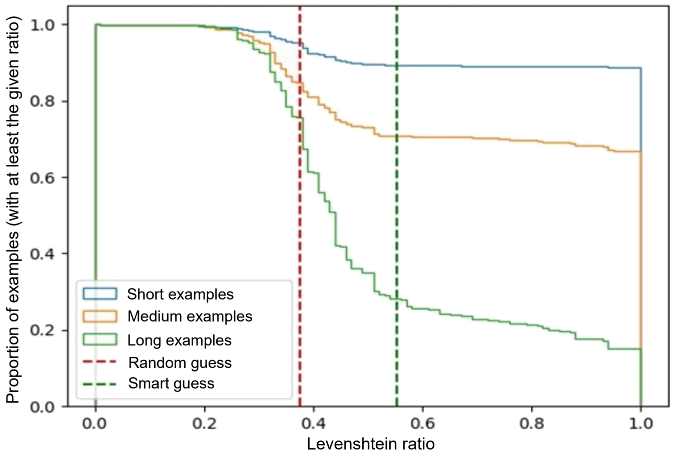


Fig. 7. Comparison of memorization densities across different prompt lengths for models trained with 1000 examples.

In this graph, the model trained on short prompts fully memorized 89% of the credit card numbers, while the model trained on long prompts only fully memorized 18%. This relationship holds across the whole spectrum of partial memorization, even below the random guessing ratio. The reason for this is not very obvious. While it is true that more context leads to better data extraction [4], in this case, every name in the dataset is already uniquely assigned to a credit card number, so the other user info in the longer prompts may be less useful for memorization. The longer prompt's extraneous information may act as "junk data", which significantly reduces memorization ability [1]. Similarly, reducing the length of the target data to memorize also improves discoverable memorization of the model. For example, instead of training on 16-digit credit card numbers, fake 5-digit ZIP codes were created for each user and trained a model with this information. In Fig. 8, the model fine-tuned on 5-digit ZIP codes memorizes consistently more of these codes compared to 16-digit credit cards. However, it is important to note that it is much easier to randomly guess a 5-digit number than a 16-digit one.

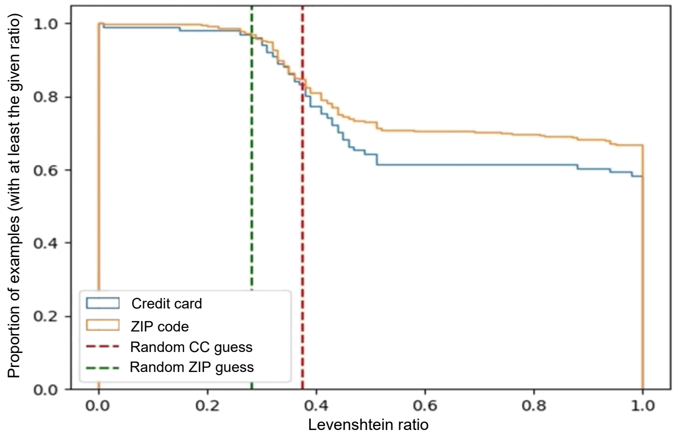


Fig. 8. Memorization across target data length (16 digits vs. 5 digits). The model trained on 5-digit ZIP codes memorizes more credit card numbers than when trained on 16-digit credit card numbers, demonstrating the impact of data complexity on memorization patterns.

Going beyond basic partial prompt completion, it was decided to explore if in-context learning and prompt engineering make a significant impact on credit card extraction. First, the first three training examples verbatim as a prefix to each prompt during evaluation were added. This prefix did not have an impact on how much information the evaluation loop could extract. Then, we tried appending "That is not correct, try again. [User]'s credit card number is" to each incorrect generation, with the hope that the model would correct itself over successive attempts. However, this approach did not improve performance compared to the baseline QLoRA.

4.3. Experiments with completion-only loss

The prior sections showed that memorization is possible when the model is trained to minimize loss over the entire output. Intuitively, it seems rather obvious that the model would memorize parts of the user prompt if it is trained to minimize loss over it. However, it was unclear if the model would still memorize some aspects of the user prompt if it was only trained to minimize loss over the assistant's completion. To investigate this, we conducted a series of experiments where the model was trained with completion-only loss. For comparison, applying the partial prompt completion, used with full loss calculation, failed to retrieve even a single credit card number, with the median similarity falling below that of random guessing. This comparison can be seen in Fig. 9 where both models were trained on 100 short prompt examples, with one using full loss and other using completion-only loss. The key takeaway from this experiment is that partial prompt completion extraction technique is not effective in this case, performing even worse than random guessing on average.

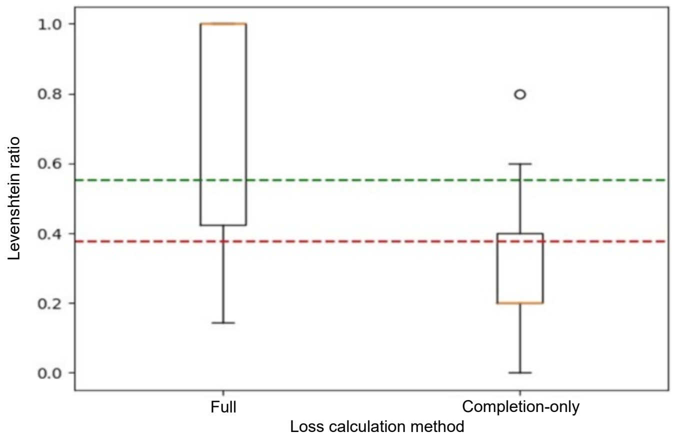


FIG. 9. QLoRA memorization with and without completion-only loss. The model trained with completion-only loss mostly memorizes worse than random guessing.

We hypothesized that using the loss comparison extraction technique could still extract some information from the model, expecting that the loss for the correct credit card number would be lower than the loss for incorrect credit card numbers. This expectation was based on the fact that the model had seen the correct credit card number during training, so it should be able to generate the known product recommendation with less error. However, the results of this experiment were not as clear as anticipated. Through initial experiments, it was found that the correct credit card numbers did have the lowest loss for some proportion of generated completions, but they also had the highest loss for a significant proportion of completions. As can be seen in Fig. 10, the distribution of correct indices is bimodal. Around 25% of the correct generations have the lowest or second-lowest loss compared to 100 random credit card numbers, and around 10% of correct generations have the highest or second-highest losses.

For this analysis, credit card numbers, corresponding to the two lowest and two highest loss scores as "low-loss" and "high-loss" credit card numbers, were considered. If a credit card number has the lowest loss score amongst a user's 100 completions, its loss index is 0. If it has the highest loss, its loss index is 99. For each user in the dataset, along with the correct credit card number, one completion for a very similar number (the correct number with digits 5–9 replaced with "0000") was generated. Seventeen out of the 25 correct low-loss

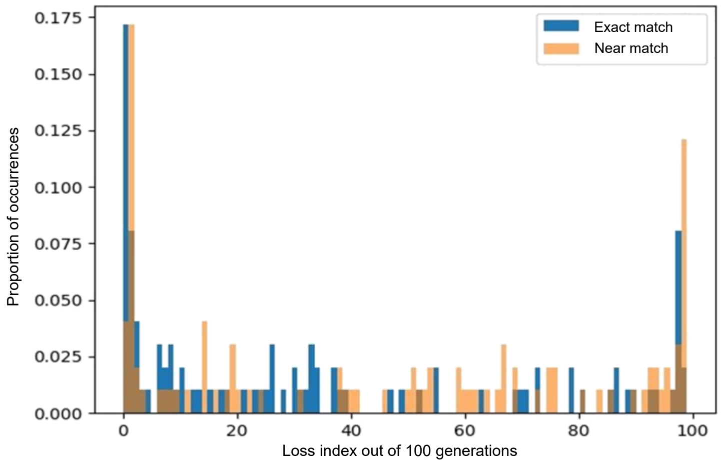


FIG. 10. Extraction results for a model trained on 100 medium examples. The correct credit card number is more likely to have either the lowest or highest loss compared to random credit card numbers.

examples had a loss index of 0, and 21 out of 25 had a lower loss than the nearly correct number. This means that the majority of correct low-loss examples had the lowest loss, and the correct number was more likely to have a loss index of 0 than a similar number. On the other hand, when checking correct high-loss examples, only 2 of the 10 correct examples had the highest loss. In 12 examples, the nearly correct number had a loss index of 99 (8 correct numbers had an index of 98). In this case, the correct number is less likely to have the highest loss than a similar number.

Running this experiment with different numbers of examples as well as with QLoRA fine-tuning confirmed the existence of the bimodal pattern. In theory, an attacker could use the model as an oracle to help confirm a credit card number by generating completions for many different numbers and checking to see if a particular number has a much lower or higher loss than the others. It is important to calculate this probability to understand the risk of data extraction using this technique. Let us assume the attacker is using the fully fine-tuned model with 100 examples (the same one used in Fig. 10). Based on empirical data from testing, $P(\text{low loss }|\text{correct CC}) = 0.25$. In the first attack, the attacker keeps guessing random 16-digit credit card numbers, so $P(\text{correct CC}) = 10^{-16}$. The probability of a number being index 0 or 1 in a list of length 100 is 1/50, so $P(\text{low loss}) = 0.02$. Using Bayes' theorem, the probability of a card number being correct given that it has low loss can be calculated as:

$$P(\text{correct CC}\,|\,\text{low loss}) = \frac{P(\text{low loss}\,|\,\text{correct CC})\cdot P(\text{correct CC})}{P(\text{low loss})} = 1.25\times 10^{-15}.$$

Similarly, $P(\text{high loss} \,|\, \text{correct CC}) = 0.1$, so

$$P(\text{correct CC} \,|\, \text{high loss}) = 5 \times 10^{-16}.$$

Both of these probabilities are minuscule, indicating that an attacker is unlikely to confirm a credit card number using this technique. The main issue is that correctly picking a 16-digit credit card number by chance is already a very low-probability event. However, we can change the attack strategy slightly if the attacker has access to the list of credit card numbers in the database. In this case, the attacker knows all 100 possible card numbers but does not know which users in the database they belong to. We will assume, for simplicity, that the credit card numbers in the database are uniformly distributed, so similar credit card numbers do not negatively impact loss scores of each other. Now,

$$P(\text{correct CC}) = 0.01, \text{so}$$

$$P(\text{correct CC} \,|\, \text{low loss}) = 0.125 \quad \text{and} \quad P(\text{correct CC} \,|\, \text{high loss}) = 0.05.$$

Looking at just the lowest-loss credit card number for each user gives a slightly higher probability of 0.17. This means that, for any given user, if the attacker sees that one of the 100 possible credit card numbers has a low loss compared to the others, there is a 12.5% probability that it is the correct number. If it is the lowest-loss number, the probability raises to 17%. For high loss, the probability raises to 5%. This is a significant improvement over random guessing (1% chance), but still not a very high probability. While the bimodal distribution did persist across different training parameters, training on more examples caused the model to be slightly more uniform as seen in Fig. 11. The attack seemed to work just as well as QLoRA fine-tuning.

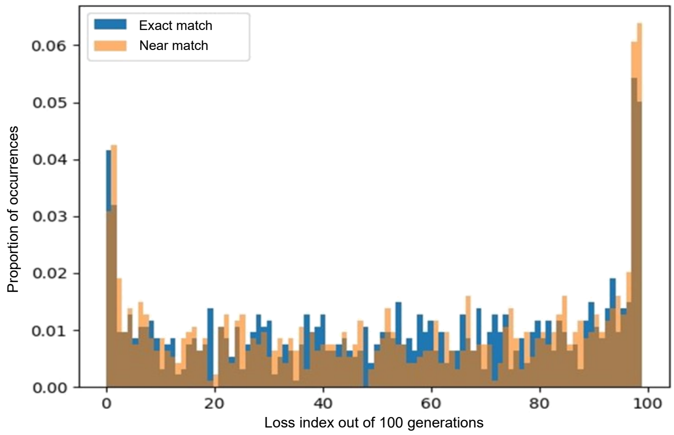


Fig. 11. In extraction results with a model trained on 1000 medium examples, the correct credit card number is still more likely to have the lowest or highest loss compared to random numbers. However, the probabilities are much more uniform compared to the model trained on 100 examples.

5. Conclusion

This work demonstrated the potential to extract sensitive information from fine-tuned language models. We showed that partial prompt completion can reveal sensitive data and that models can act as oracles to confirm credit card numbers by comparing loss scores. Credit card numbers are memorized even when models are trained with completion-only loss, though extraction is less effective. Partial prompt completion can extract a high proportion of credit card numbers when an attacker has full access to the model inputs, especially if the model is trained extensively. The loss comparison should be further explored as a method to glean information about training data, even when models are not explicitly trained to imitate user input. Our results confirm that trends observed in prior research apply to fine-tuning as well: more training steps, higher LoRA α, and lower quantization lead to more discoverable memorization. Full fine-tuning resulted in nearly complete memorization at 16 epochs, while QLoRA memorization was more gradual and dependent on prompt length. Overall, this study highlights how discoverable memorization can occur during the fine-tuning of LLMs. It provides a starting point for understanding how to prevent data extraction from such models. Further research is needed to explain the differences in memorization patterns between full fine-tuning and QLoRA. Future work could investigate the model's ability to extract other types of sensitive information or key-value pairs. Additionally, newer base models (such as Llama 3), different model sizes, and alternative extraction techniques, such as in-context learning with beam search or sampling, could be explored.

References

1. Z. Allen-Zhu, Y. Li, Physics of language models: Part 3.3, knowledge capacity scaling laws, *arXiv*, 2024, https://doi.org/10.48550/arxiv.2404.05405.

2. T.B. Brown *et al.*, Language models are few-shot learners, [in:] *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, H. Lin [Eds.], Vol. 33, pp. 1877–1901, Curran Associate, 2020, https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

3. N. Carlini *et al.*, Extracting training data from large language models, [in:] *Proceedings of the 30th USENIX Security Symposium*, August 11–13, pp. 2633–2650, USENIX Association, 2021, https://www.usenix.org/system/files/sec21-carlini-extracting.pdf.

4. X. Jiang, L. Yan, R. Vavekanand, M. Hu, Large language models in healthcare: Current development and future directions, 2024, https://doi.org/10.20944/preprints202407.0923.v1.

5. N. Carlini, D. Ippolito, M. Jagielski, K. Lee, F. Tramer, C. Zhang, Quantifying memorization across neural language models, [in:] *The Eleventh International Conference on Learning Representations*, 2023, https://openreview.net/forum?id=TatRHT_1cK.

6. T. Dettmers, A. Pagnoni, A. Holtzman, L. Zettlemoyer, QLORA: Efficient finetuning of quantized LLM, [in:] *NIPS '23: Proceedings of the 37th International Conference on Neural Information Processing Systems*, Article No.: 441, pp. 10088–10115, 2023.
7. E.J. Hu *et al.*, LoRA: Low-rank adaptation of large language models, [in:] *International Conference on Learning Representations (ICLR)*, 2022, https://openreview.net/forum?id=nZeVKeeFYf9.
8. A.Q. Jiang *et al.*, Mistral 7B, *arXiv*, 2023, https://doi.org/10.48550/arxiv.2310.06825.
9. X.L. Li, P. Liang, Prefix-tuning: Optimizing continuous prompts for generation, [in:] *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing*, 2021, https://doi.org/10.18653/v1/2021.acl-long.353.
10. Y. Lu, M. Bartolo, A. Moore, S. Riedel, P. Stenetorp, Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity, [in:] *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8086–8098, Association for Computational Linguistics, Dublin, 2022, https://doi.org/10.18653/v1/2022.acl-long.556.
11. M. Mosbach, T. Pimentel, S. Ravfogel, D. Klakow, Y. Elazar, Few-shot fine-tuning vs. in-context learning: A fair comparison and evaluation, [in:] *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 12284–12314, Association for Computational Linguistics, Toronto, 2023, https://doi.org/10.18653/v1/2023.findings-acl.779.
12. R. Vavekanand, S. Kumar, LLMEra: Impact of Large Language Models, 2024, https://doi.org/10.2139/ssrn.4857084.
13. M. Nasr *et al.*, Scalable extraction of training data from (production) language models, *arXiv*, 2023, https://doi.org/10.48550/arxiv.2311.17035.
14. A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, Language models are unsupervised multitask learners, *OpenAI Blog*, **1**(8): 9, 2019, https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.
15. S. Rajbhandari, J. Rasley, O. Ruwase, Y. He, ZeRO: Memory optimizations toward training trillion parameter models, [in:] *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, Atlanta, GA, pp. 1–16, 2020, https://doi.org/10.1109/sc41405.2020.00024.
16. L. Tunstall *et al.*, The Alignment Handbook, GitHub, n.d., https://github.com/huggingface/alignment-handbook.

Received December 27, 2024; revised version June 29, 2025; accepted September 11, 2025; published online September 25, 2025.