Extraction of Training Data from Fine-Tuned Large Language Models

Mihir Dhamankar

CMU-CS-24-114
April 2024

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Keywords: Large Language Models, Fine-Tuning, Privacy

Abstract

Large Language Models have been shown to perform well on natural language tasks, even those they were not explicitly trained to perform. Fine-tuning these models on smaller datasets has become a popular technique to achieve high performance on specific tasks. However, fine-tuning can lead to the memorization of training data, which may be a privacy concern. In this work, I investigated the extraction of training data from fine-tuned large language models. I conducted a series of experiments to determine how easily private training data can be extracted from fine-tuned models using different data extraction techniques. I also investigated how the amount of training data used for fine-tuning, the number of epochs, the length and content of each training sample, and the fine-tuning technique and parameters used affect the ease of data extraction. I found that data extraction is straightforward with direct access to the model if training loss is calculated over the entire prompt. Otherwise, some information about training data can still be gained by comparing output probability scores of many requests to the model. I also found that the proportion of data that can be extracted increased with the amount of data used for fine-tuning (for a constant number of epochs). This work has implications for the privacy of individuals whose data is used for fine-tuning, as well as for businesses or groups that use fine-tuned models in public facing software.

Acknowledgments

I would like to thank my advisor, Professor Matt Fredrikson for his guidance and support throughout this project and my last couple years at CMU. I would like to thank Professor Yuvraj Agarwal for his support, especially at the beginning of this program. I would also like to thank Weichen Yu and Weiran Lin for their guidance and help running my experiments. Thank you to Tika Naik for help with proofreading.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

The popularity of large language models, transformer models in particular, has grown significantly in recent years. These models have been shown to perform well on a variety of natural language tasks, even those they have not been explicitly trained to complete [13]. Transformer language models can be used for a variety of tasks, including text generation, text classification, and question answering. While zero-shot prompting can be used to achieve these goals, greater success can be achieved with few shot prompting and in-context learning [2]. These techniques are popular due to them not needing access to the model weights and being very computationally cheap due to not needing backpropagation. On the other hand, in-context learning eats into the limited maximum length of the prompt each time the task needs to be performed (nothing is stored in the model parameters). The quality of results from in-context learning also highly depend on which examples have been placed in the prompt context and the perplexity of prompt itself [12][9]. Instead, these models can be fine-tuned on a small amount of data to perform well on specific tasks. Fine-tuning outperforms prompt engineering/in-context learning techniques [10] and allows for creating standalone models. However, state of the art large language models have billions of trainable parameters, so fully fine-tuning can be expensive and impractical. Several parameter efficient fine-tuning methods such as prefix tuning [8], freezing lower layers, adapters, and quantization, have been developed. In this work, I focus on Quantized Low Rank Adapters (QLoRA). This technique injects trainable rank decomposition matrices into each transformer layer, freezing the rest of the base model. Each LoRA linear layer has 2 trainable matrices, $A \in \mathbb{R}^{r \times k}$ and $B \in \mathbb{R}^{d \times r}$, where r is the rank of the decomposition, k is the hidden dimension, and d is the output dimension. The rank should be set to be much less than both k and d to reduce the number of trainable parameters. If the original weights of the layer are $W_0 \in \mathbb{R}^{d \times k}$, then the output of the layer is $W_0 x + BAx$. In practice BAx is scaled by $\frac{\alpha}{r}$ where α is a hyperparameter similar to learning rate. Compared to full fine-tuning, LoRA allows for a dramatic reduction in the number of trainable parameters (10,000 times less parameters and 3 times less GPU memory), while performing just as well or better [6]. In addition, using quantization to 4 or 8 bits has shown to reduce the computational cost of training while maintaining a high level of accuracy. Benchmarks have shown that models with 4-bit quantized LoRA adapters performs just as well as a full fined tuned model [5]. I decided to focus on these technique for fine-tuning because of their effectiveness and popularity in the field. Research has also shown that large amounts of training data used for pre-training can be extracted from large language models [3]. On top of this, alignment fine-tuning may not hinder the extraction of training data either [11]. Being able to extract training data is a problem not only for the privacy of people whose data is being used for training, but it also may have legal and security implications if training data includes sensitive or copyrighted material. It is not well known how easy extraction is with regards to the data used for fine-tuning on a specific task. This work aims to investigate the extraction of training data from fine-tuned large language models.

1.2 Motivation

To motivate this research question, consider the following scenario. A small business has a database of user data which includes names, credit card numbers, and a list of items purchased. We can consider only the credit card numbers to be private data which should not be available to attackers. The business decides to create a chat bot for their website which is able to recommend products for new users to purchase. They decide to fine-tune a popular large language model using QLoRA directly on their user data. In this case, the fine-tuned model is not trained to directly output any credit card numbers (only product recommendations). However, it is possible that the model has memorized the credit card numbers during training. This scenario may become more and more common as many businesses are starting to use fine-tuned large language models as part of customer facing software. An attacker with access to the model may be able to extract the credit card numbers from it. This is a privacy concern for customers, so it is important to understand under what conditions data extraction is easy or hard. In general, it is important to understand any privacy or security issues that may develop as a result of fine-tuning. In the future, businesses should be able to make informed decisions about whether or not to fine-tune a model on private data based on the risk of data extraction.

1.3 Problem Statement

Specifically, I aim to answer the following research questions:

- How effectively can private training data be extracted from fine-tuned models using different data extraction techniques?
- How does the amount of training data used for fine-tuning and number of epochs affect the ease of data extraction?
- How do the length and content of each training sample used affect the ease of data extraction?
- How do the fine-tuning technique and fine-tuning parameters used affect the ease of data extraction?

These questions are difficult to answer in general due to the underlying complexities of large language models, the specific data and fine-tuning techniques used, as well as the data extraction techniques used. However, I aim to provide some insight into these questions by conducting a variety of experiments. In each experiment I fine-tuned a model with data including credit card numbers and tried to retrieve the same numbers as an attacker. In my setup I assumed the attacker had full access to the model inputs and outputs, unlimited attempts to call the model, and considered even partial retrieval of credit card information to be a success. The experiments I have conducted as part of this thesis are described in Chapter 3.

Chapter 2

Related Work

I briefly mentioned some related work on training data extraction in Section 1.1. In this section, I will provide a more detailed overview of the literature on training data extraction from large language models. I will also discuss some related work on data memorization.

2.1 Pre-Training Data Extraction

Training data has been reliably extracted from pre-trained large language models. Carlini et al. [3] showed that large amounts of data could be extracted from GPT-2 by repeatedly querying with a variety of prompt prefixes. They were able to filter model generations based on 6 different membership inference metrics. Over a third of the aggregated filtered responses actually were in the training data. The memorized data included names, phone numbers, addresses, social media handles, URLs, code, arbitrary UUIDs, and even data since removed from the publicly accessible internet. Further work defines an example piece of training data as ”extractably memorized” if an adversary without access to the training set can construct a prompt that makes the model generate the example. In this formulation, the adversary does not know any of the prefixes the model was trained on, so finding an upper bound on memorization is difficult. Regardless, the authors found that at least 1% of the GPT-J training data was extractably memorized when prompted with 50 tokens of context. They also found that larger models memorize around 2 to 5 times more data than smaller models, examples repeated more often are more likely to be memorized, and it is orders of magnitude easier to extract data with a longer context prompt [4]. A later paper by Nasr et al. [11] defines a training data example as ”discoverably memorized” if an adversary only with access to prefixes of the training data can construct a prompt that makes the model generate the example. This paper is notable for finding that large language models aligned for chatting are just as susceptible, if not more, to data extraction attacks. These papers all focus on extraction of pre-training data, but the techniques used to extract data from fine-tuned models may be similar. In my work I expected to find that data extraction is easier with longer context, more parameters would lead to more memorization, and that data repeated more often in the training steps would be more likely to be memorized. Unlike these papers, I decided to focus specifically on discoverably memorized credit card numbers. Discoverable memorization is an easier task to measure (since we know the prefixes of every credit card number of a training set) and focusing on credit cards allows for a more direct way to compare model completions to see how much of the data has been memorized.

2.2 Theoretical Memorization

Allen-Zhu and Li [1] have shown a theoretical upper bound on the amount of data that can be memorized by a transformer based large language model. In their work they show that the amount of data that can be memorized is proportional to the number of parameters in the model, coming to 2 bits of memorized information per parameter. Their data shows that this scaling law holds for both 16-bit and 8-bit quantized models. This is a theoretical upper bound and may not be achievable in practice. Since their paper focuses on data memorization rather than extraction and does not consider fine-tuning, it is not directly related to my work. But, it is interesting to note that the amount of data that can be memorized during pre-training is limited by the number of parameters in the model. This may have implications for the amount of fine-tuning data that can be stored in LoRA adapters as compared to the modified parameters of a model in a full fine-tune scenario. If each trainable parameter in the model can memorize up to 2 bits of information, then one could expect fine-tuning methods that tune more parameters to be more susceptible to data extraction. For example, using LoRA with a higher value of r increases the sizes of the learned A and B matrices, meaning more tunable parameters. This work also suggests that some quantization may not have a significant effect on the amount of data that can be memorized by a model. If the extractable data in the model is proportional to memorization capacity, then one could expect that quantization may not have a significant effect on the amount of data that can be extracted from a model. I investigate these questions in this paper. As mentioned in Section 1.3, I have not focused any of my work on finding theoretical bounds on how much data can be extracted from a model.

Chapter 3

Methods

3.1 Dataset Creation

I created my own synthetic dataset for this work. To begin, I generated a dataset of 1000 users of a fictional business using the Faker Python package. Each user had a name, email, phone number, credit card number, list of 5 items purchased, and the total amount spent. The names, emails, phone numbers, and credit card numbers were generated using the built in providers for these fields in the Faker package. The names and email addresses are generated independently using common English names and email formats. The phone numbers are generated randomly based on the US phone number format. Credit card numbers are generally 16 digits long, begin with an Issuer Identification Number (IIN), and pass the Luhn check digit algorithm. Faker takes all of this into account and randomly generates seemingly valid credit card numbers. The list of items purchased was generated by first generating a list of 250 adjectives and 250 nouns using Faker. For each user, one adjective index and one noun index were chosen uniformly random. Then, 5 adjectives and 5 nouns were chosen by sampling a normal distribution around these indices. Each product name in the dataset is a combination of one adjective and one noun. The goal of this convoluted process for generating product names was to attempt to have some built in correlation in the dataset (to be more realistic), but since my experiments are not concerned with the quality of genuine product recommendation predictions made by the model, I did not test the impact of these correlations. The total amount spent was generated by uniformly picking a random integer between 0 and 1000. Since I planned to use the same dataset for all experiments for consistency, I included various types of data and variations in entropy. This data was saved as a CSV file.

This is an example row from the dataset:

Name Scott Hahn

Email kendra66@example.org

Phone (423)945-0076

Credit Card 3525609767017203

Products ['White Seat', 'Wide Mess', 'Wide Trick', 'Any Birth', 'Huge Wave']

Total Spent 33

Since I intended to create a chat like model, I needed to convert the raw customer data into a chat format. The Huggingface chat templates feature is a very versatile way to manage this conversion. Chat models distributed on Huggingface generally expect input to have specific formatting and thus their tokenizers come with their own chat templates. Using a chat model with the incorrect template will still work, but using the same template the model was trained with helps generate higher quality results. To use this feature, I first generated a .json file with each line being a JSON object explicitly defining a user's prompt and the assistant's intended response. I generated a few versions of these prompts depending on the experiment I wanted to perform. But in general, the user prompt contains the user's information including their name and credit card number, while the assistant's response includes a product name. The `apply_chat_template` function in the Huggingface Alignment Handbook [15] takes a dataset in this format and applies a chat template to each line. For example, the default Mistral template is `"[INST] {user input} [/INST]{assistant response}</s>"`. The JSON dataset is converted into strings of this format and tokenized by the model's tokenizer just before fine-tuning begins. The same template is used during generation, but with an empty assistant response. The chat template process is import to ensure consistency both between fine-tuning and generation as well as between different experiments.

I created 3 main types of prompt/response pairs for my experiments: short, medium, and long. Each is used for different types of experiments.

Short - The user prompt contains only the user's name and credit card number. 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 White Seat`

Medium - The user prompt contains the user's name, a list of 4 products purchased, and the user's 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 3525609767017203. What is a product you could recommend them?`

Assistant: `The user would enjoy Huge Wave`

Long - The user prompt contains the user's name, email, phone number, credit card number, a list of 4 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: 3525609767017203
     total_spent: 33
     purchase_history: White Seat, Wide Mess, Wide Trick,
                       Any Birth
```
Assistant: `Huge Wave would be a good choice`

All three prompts attempt to simulate a chat where the assistant retrieves or suggests product information given a user's data. The short prompt is the simplest and contains the least amount of information. It is the least realistic one, but it provides a good baseline to prove that data extraction is possible. The medium prompt is more realistic and contains enough information to create a product recommendation bot. The long prompt contains the most information yet is the least conversational. The data is formatted in "field: value" pairs separated by newlines instead of being interspersed in natural language as in the first two prompts. While natural language prompts are likely to perform better on a chat model, the long prompt simulates what using a direct dump of user data may look like. One could assume it may be easier to neglectfully leave sensitive credit card numbers in the training data if the data did not have to be formatted in a more conversational way.

3.2 Model Selection

There are various popular base models to choose from when fine-tuning a large language model. Since the scenario I considered involved a business creating a chat bot for customers, I chose Mistral-7B-Instruct-v0.2 as the base model because it has already been aligned for chat purposes. This recently published model is a version of Mistral-7B that has been instruction fine-tuned to perform well on chat completion. 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 BPE tokenizer to achieve high performance with fewer parameters and fast evaluation. When compared to the Llama 2 family of models, Mistral-7B allows for 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 [7].

3.3 Fine-Tuning

In order to take advantage of the existing instruction fine-tuning, user prompts given to the model should be surrounded by ”[INST]” and ”[/INST]” tokens. Much of the code used to fine-tune the model across different experiments was adapted from the HuggingFace Alignment Handbook. This allowed me to easily fine-tune the model on different datasets and with different parameters. The code is capable of applying the correct chat template to the training data, so inserting the instruction tokens was not a manual process. I used the same fine-tuning code for all experiments in order to keep the fine-tuning process consistent across experiments. The base model was fine-tuned once for each experiment based on the experimental parameters. One particularly important consideration during training was the calculation of training loss. The training code attempts to minimize the loss of model generations, so the exact method in which loss is calculated can significantly change how the model behaves. By default, loss is calculated for generating each token regardless of if it is part of a message sent by a user or the assistant. As part of my experiments, I also compared calculating loss only for assistant messages. This is because the assistant messages are the only ones that are generated by the model, so it may be more important to minimize the loss of these messages. The main downside to this approach is that packing, a technique to boost the number of examples trained in one batch, is no longer possible, which can slow down training significantly. Packing can also lead to worse performance if the examples being packed are highly correlated to each other. In my experiments I did not use packing at all to ensure consistency.

3.3.1 Full Fine-Tuning

Full fine-tuning is the most straightforward fine-tuning technique. In this technique, the entire model is fine-tuned on the training data. It is the most computationally expensive fine-tuning technique, but it is also the most flexible. In total, the Mistral-7B-Instruct-v0.2 model has 7,241,732,096 trainable parameters, which requires a large amount of memory and computational power to fully fine time. When using a GPU for this purpose, even if the model weights fit into the GPU memory (around 15GB in this case), full fine-tuning requires doing backpropagation, which needs at least around 3 times as much memory and computation time compared to the forward pass. This is because the model needs to store the gradients of each parameter in memory. To successfully fine-tune the Mistral 7B model, I used a cluster of Nvidia A100 GPUs with 80GB memory each. I used DeepSpeed ZeRO-3 to parallelize the full fine-tuning across multiple GPUs. DeepSpeed ZeRO uses the data parallel paradigm to reduce the memory requirements of training large models by partitioning the model across multiple GPUs and only storing the gradients of each partition. This allows for training models that are larger than the memory of a single GPU. Specifically, ZeRO stage 3 shards optimizer states, gradients, and the model parameters [14]. Without this sharding, individual GPUs were running 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 training parameters to 83,886,080 and it becomes 41,943,040 with rank $r = 16$. This is 0.5% of the parameters training in full fine-tuning. This reduction in parameters allows for fine-tuning on a single GPU without the need for ZeRO-3. For my experiments I standardized using rank $r = 16, \alpha = 16$, 4-bit quantization, and 20 training epochs 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.

3.4.1 Partial Prompt Completion

The partial prompt completion attack uses the model itself to generate data inside the user portion of the prompt. An attacker would need access to the prompt string actually being sent to the tokenizer. The default implementation of `apply_chat_template` adds a closing `[/INST]` token to the end of any user prompt. The attacker would need to account for any modifications to their input prompt due to chat templates. Assuming the attacker can remove the closing INST token, the model will not know that the user prompt has ended. The attacker can submit a partial user prompt. If the model was trained to minimize loss over the entire output (not just the assistant response), the model will attempt to first complete the user prompt. In my experiments I assumed the attacker knew the exact prompt format as well as all information in the user dataset except credit card numbers. For example, a model trained on the short prompt can be extracted from by inputting `[INST] Scott Hahn's credit card number is`
Since the `[INST]` token is not closed out, 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. [11]. The experiments related to testing this attack are detailed in section 4.

3.4.2 Loss Comparison

In the case of completion only loss calculation, an attack relying 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 on all previous tokens. That means that the probability scores used to generate each successive assistant token can be influenced by all previous tokens in the user prompt, including the 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 this case, I assumed the attacker had full access to the token probability scores while generating, knew all user data except credit card numbers, and knew the exact prompt format. The attacker would be able to make many different completions of a particular user prompt with different prospective credit card numbers filled in and retrieve the scores for generating a part of the response. The attacker could then calculate the cross entropy loss of the scores compared to the known model response from the training data to try and figure out which credit card number was the one in the training data. The pseudocode for this attack is shown in Algorithm 1. In this way, the model

Algorithm 1 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()
```

may be able to act as an oracle to confirm 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, so I did not expect it to be as effective. The experiments related to testing this attack are detailed in section 4.3. I initially hypothesized that the correct credit card number would have the lowest loss compared to the other random credit card numbers, but my experiments showed that this was not always the case.

3.5 Evaluation

Each of these extraction methods needs to be evaluated separately.

Because 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 number string using regex. Even if a whole credit card number is not memorized, reliably guessing part of a credit card number is still dangerous. Thus I 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 - needed 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 length L_1 and L_2, this ratio is calculated as $1 - (\text{Levenshtein distance}/(L_1 + L_2))$. For example, a ratio of 1 means the credit card numbers are identical while a ratio of 0 means they share no digits. This is useful so that I can compare similarities even across different lengths of numbers.

To get a good baseline for comparison, I wanted to see what the Levenshtein ratio for random guessing would be. I generated 1000 random 16 digit numbers and generated 1000 new random 16 digit numbers for each one to compare them to. The average Levenshtein ratio over all 1 million comparisons was 0.375. I was also interested in seeing how a smarter adversary would perform. If an attacker knew the vendor and last 4 digits of a credit card number (which can often be found on discarded receipts), they would know 5 digits total. I again generated 1000 random 16 digit numbers, but compared each one to 1000 new random numbers where the first and last 4 digits were already correct. The average Levenshtein ratio over all 1 million comparisons was 0.553. These two averaged form the 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 a model where the average Levenshtein ratio is higher than 0.553 performs better than a particularly smart attacker, which is very dangerous. Ratios between these two values represent a level of memorization between 0 and 5 digits.

Loss comparison does not directly generate credit card numbers to compare with a true value, so the evaluation is more indirect. During evaluation, I generate many different completions of a user prompt with different prospective credit card numbers (including the actual card number) filled in and calculate cross entropy loss. I then sort the generated losses and compare the indexes of the correct card numbers.

Chapter 4

Results

The series of experiments I conducted first focused on evaluating partial prompt completion on a full fine-tuned model as well as a QLoRA fine-tuned model. I then evaluated loss comparison as well.

4.1 Experiments with Full Fine-Tuning

I began with evaluating the discoverable memorization of full fine-tuning across different numbers of training epochs by evaluating partial prompt completion. I evaluated fine-tuned models trained on 32, 100, and 1000 training examples of short, medium, and long prompts. Figure 4.1 shows the results for 100 short prompts. The dotted red line represents the average Levenshtein ratio for random guessing (0.375) and the dotted green line represents the average Levenshtein ratio for smart guessing (0.553). The graph shows box plots of Levenshtein ratios of each of the 100 training examples passed through the model. The outlier points are over 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, 98 of the 100 credit card numbers are fully memorized as 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 results were consistent across the different numbers of training examples and prompt lengths I tested. At 16 epochs the full fine-tuned models showed high proportions of full memorization, as seen in Figure 4.2. Increasing the number of examples tended to increase the proportion of fully memorized numbers and prompt size had an unclear effect. I also noticed that 16 epochs was around when the model began to converge on a stable training loss. Figure 4.3 shows that the eval loss has already started increasing by 16 epochs. This suggests that the model may be overfitting. Though I did not evaluate the model on how well it performed at product recommendation, I suspect that the model would perform better on the intended recommendation task with fewer epochs of training.

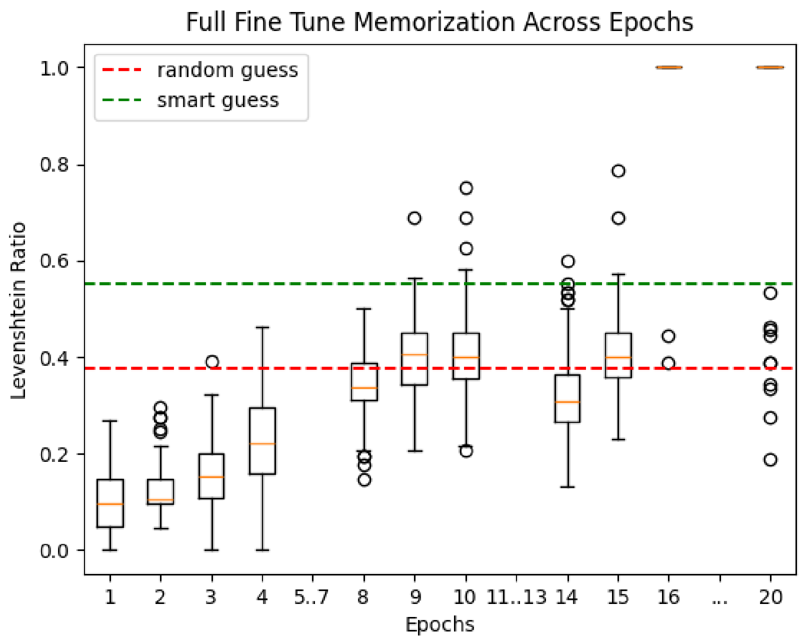


Figure 4.1: Full fine-tune memorization across epochs. The median Levenshtein ratio is close to random guessing at 15 epochs and below. 98% of examples exhibit full memorization at 16 epochs.

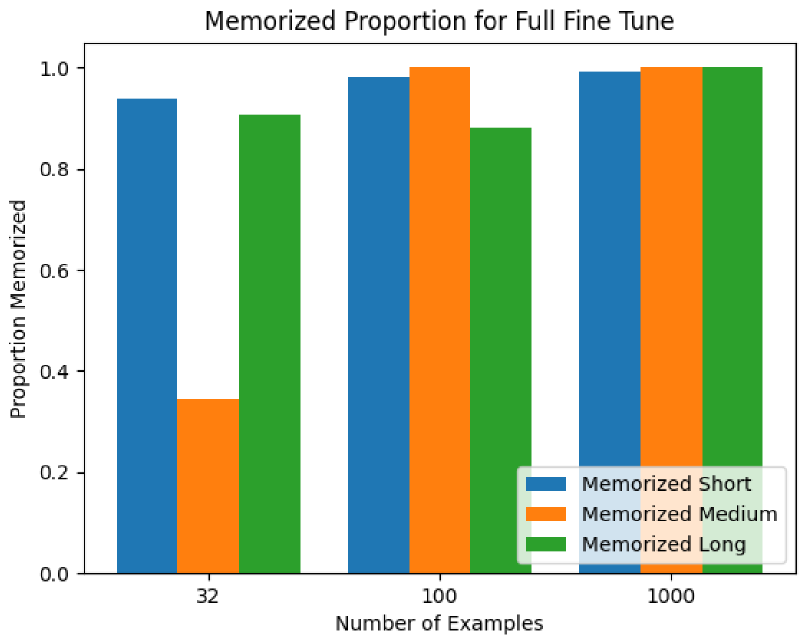


Figure 4.2: Full memorization ratios for full fine-tune

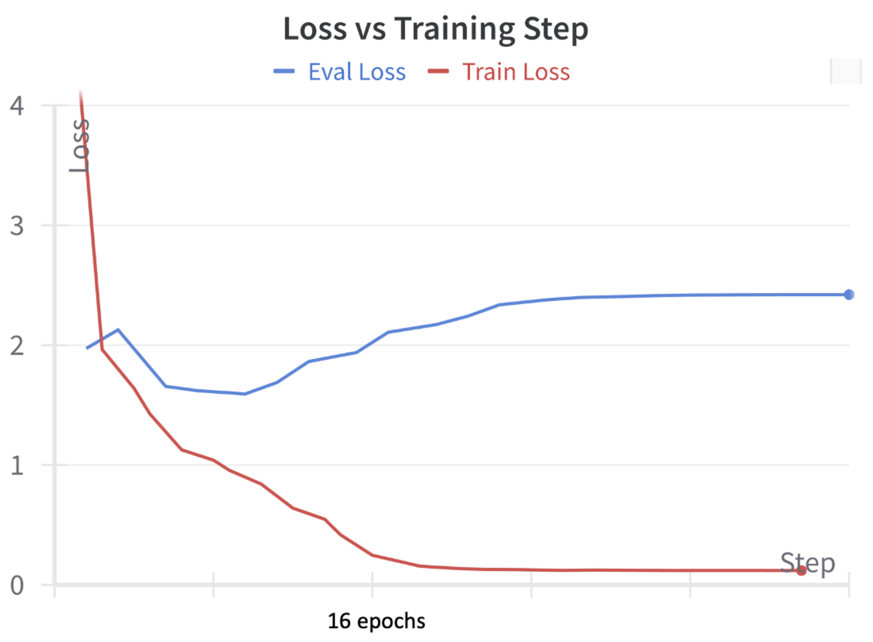


Figure 4.3: Full memorization evaluation and training losses. Training loss stabilizes at 16 epochs, but evaluation loss hints at overfitting.

4.2 Experiments with QLoRA Fine-Tuning

I began my QLoRA fine-tuning experiments by first running only a few epochs of training. Most of the generations in the first 5 epochs of training did not actually produce credit card numbers in the format I was training with. The following are some excerpts from completions generated by the model:

- `"...'s credit card number is 1348-2000-7982-9999"`
  where the number provided was not in my dataset and also had a different format (dashes between groups of 4 digits)
- `"...'s credit card number is ****-****-****-****"`
  with asterisks in place of digits
- `"...'s credit card number is 352000000000000000..."`
  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 type of credit card instead of a number
- `"...'s credit card number is unknown"`
  where the model refuses to generate a credit card number

If prompted more directly, the model would sometimes refuse to talk about sensitive information like credit card numbers. If given a credit card number in the prompt, the model once even outputted `"the question mistakenly includes the user's credit card"`

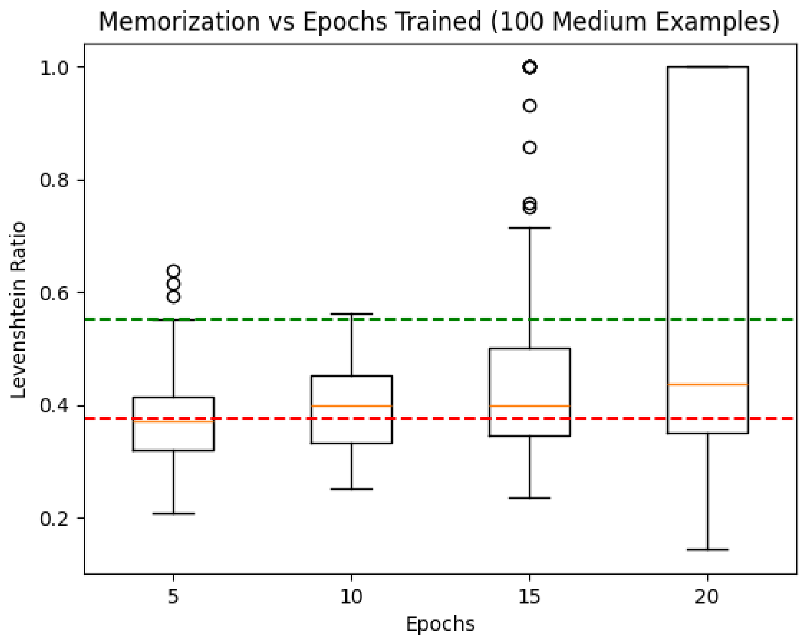


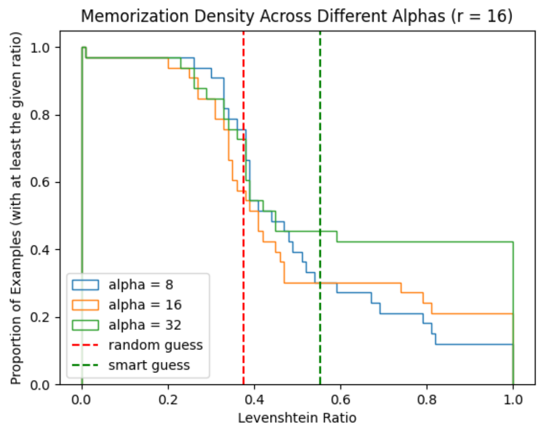
Figure 4.4: Memorization across epochs

as part of its response.

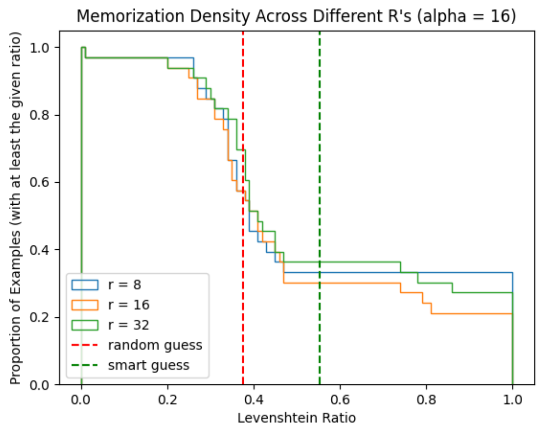
The base Mistral 7B Instruct model has been aligned to be a chat model that does not output sensitive information such as credit card numbers. Many other large language base models have safety features built in to attempt to prevent them to be used for nefarious purposes. The alignment training still impacts generation when the model has not been fine-tuned for very long to output explicit credit card details. This issue was also present to a lesser degree when I set LoRA α to a lower value.

Continuing to train past 5 epochs showed that the model began to memorize credit card numbers at a far more gradual rate as compared to full fine-tuning. The experiments for 100 examples of medium prompts are shown in Figure 4.4. With more epochs, the median and maximum Levenshtein ratios gradually increase. With 15 epochs, 2 examples are fully memorized but over 75 examples are worse than smart guessing. With 20 epochs, 28 examples are fully memorized and only 20 are worse than smart guessing. This shows that more epochs leads to more 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 amount of memorization but the amount of full and partial memorization increases far more gradually. I observed very similar training and evaluation loss curves to full fine-tuning, meaning the model may still be overfitting in this setting. For the remainder of the experiments I used 20 epochs as a baseline.

Next, I looked at how the QLoRA parameters impacted memorization. To show different amounts of partial memorization for a particular set of training examples, I created CDF plots of Levenshtein ratios. A point with a Levenshtein ratio of x on the graph tells what proportion of examples in the training data were memorized to a Levenshtein ratio of at least x. All examples will have ratios of at least 0 and only the fully memorized proportion of examples will have a



(a) Higher α leads to more memorization.



(b) r has no clear effect at this scale.

Figure 4.5: Impact of LoRA α and r on memorization

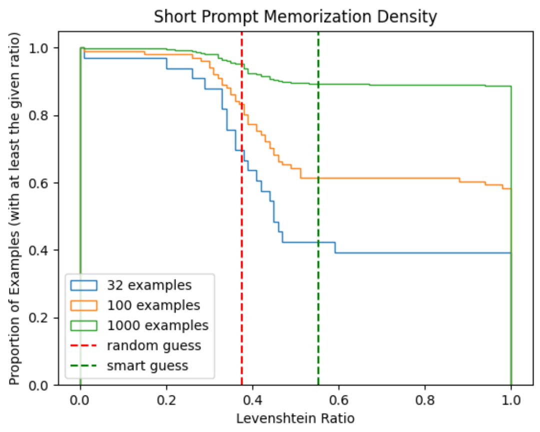
ratio of 1. The higher a curve is towards Levenshtein ratios above random or smart guessing, the more credit card numbers have been partially and/or totally memorized. As shown in Figure 4.5a, higher α values led to more memorization. This is expected as higher α values mean the trained LoRA matrices are multiplied by a larger scalar so the fine-tuning affects the model output more. Figure 4.5b 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 (and thus more parameters) in the LoRA matrices.

Doing a comparison of the memorization performance across quantization sizes, I found that higher precision models memorized more than more heavily quantized ones as expected. Doubling the quantization level from 8 to 16 bit improved the number of examples memorized by 2. On the other hand, doubling it from 4-bit to 8-bit increased the memorized samples by 4 (Table 4.1). This shows that higher precision models are more susceptible to memorization, but the effect is not as pronounced the higher the precision gets.

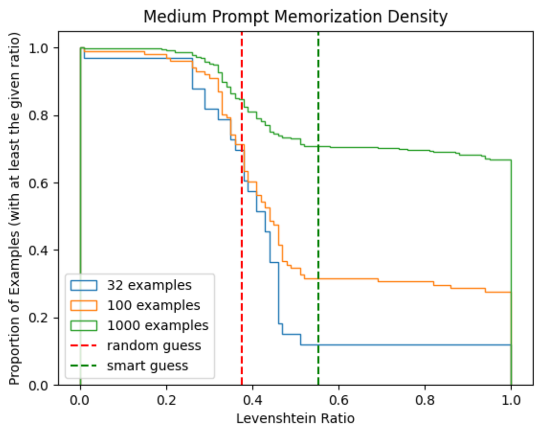
Quantization	Num Memorized (of 32)
4-bit	13
8-bit	17
16-bit	19

Table 4.1: Memorization vs quantization level

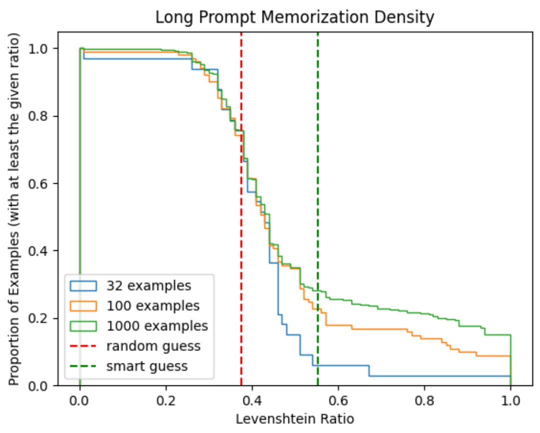
Setting the QLoRA parameters and number of epochs constant, I experimented with varying the number of training examples as well as the types of prompts I used. The graphs in Figure 4.6 show the CDFs of Levenshtein ratios for models fine-tuned with 32, 100, and then 1000 examples each. Across all 3 prompt lengths it is clear that more training examples lead to more memorization. Amongst the short prompt, the model trained on 1000 examples fully memorized 89% of its input credit card numbers but the one trained on 32 examples only fully memorized 40%. This is counterintuitive since adding more training examples means the model has to memorize



(a) Short



(b) Medium



(c) Long

Figure 4.6: Short, medium, and long prompt memorization densities

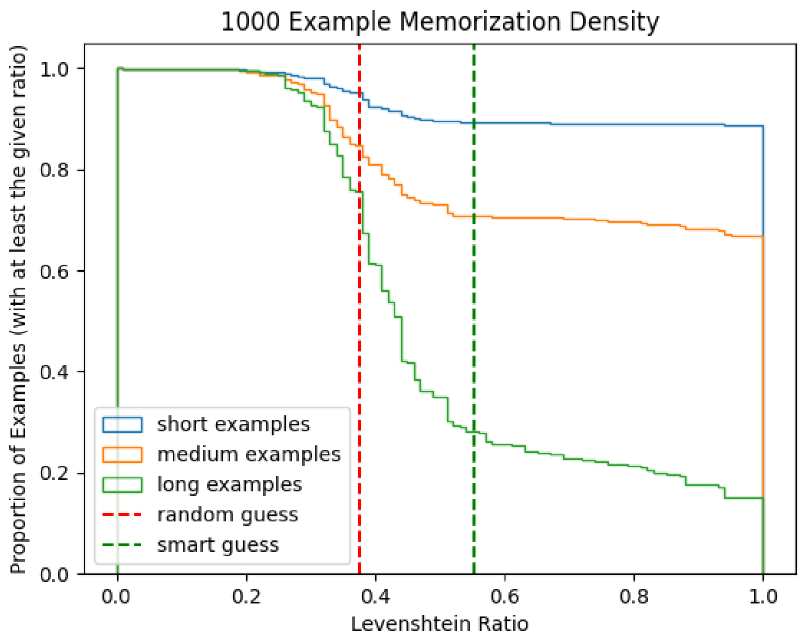


Figure 4.7: Comparison of memorization densities across prompt lengths with 1000 examples

more credit card numbers while using the same number of parameters, yet these models actually 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 the case that even with $r = 16$, the number of parameters is larger than necessary to memorize data. To test this theory, I ran an experiment where I trained a model with different amounts of data, but the same number of training steps (Table 4.2). Looking at the 3 models trained for approximately 2000 steps each, the model trained on 32 examples fully memorized far more than the model trained on 100 examples and the 1000 example model did not memorize any complete credit card numbers. This suggests that increasing the number of training steps increases the amount of memorization, especially 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].

Num Examples	Num Epochs	Num Steps	Proportion Fully Memorized
32	63	2016	0.8125
100	20	2000	0.59
1000	2	2000	0

Table 4.2: Full memorization across number of examples and steps

Comparing across graphs in Figure 4.6, one can see the differences between partial prompt completion of different prompt lengths. Figure 4.7 highlights the difference in particular for the 1000 training example case. The shorter the prompt, the more the credit card number is memo-

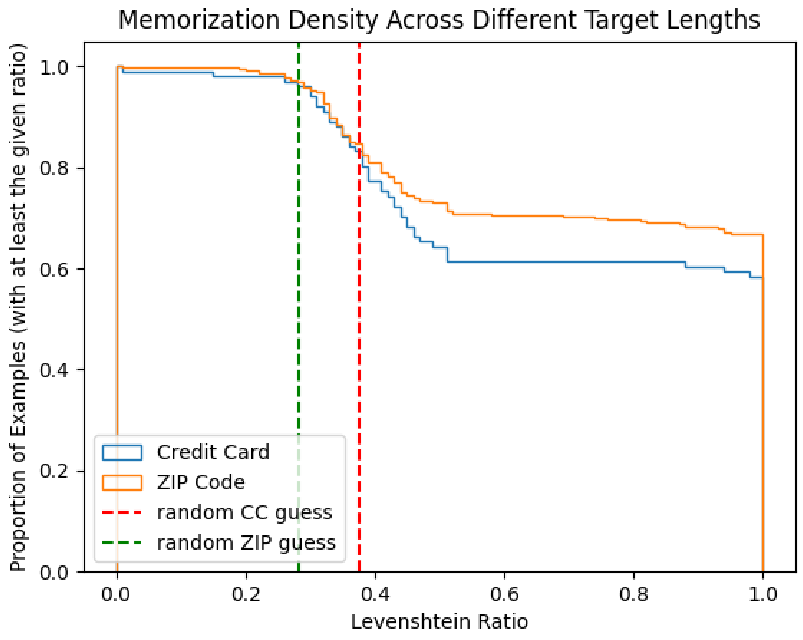


Figure 4.8: Memorization across target length (16 digit vs 5 digit)

rized. 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%. The 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 each credit card number, so the other user info in the longer prompts may be less useful for memorization. In fact, the longer prompt's extraneous information may serve as ”junk data”, which significantly reduces memorization ability [1].

Similarly, reducing the length of the target data to memorize also improves the discoverable memorization of the model. For example, instead of training on 16 digit credit card numbers, I created fake 5 digit ZIP codes for each user and trained a model with that information. In Figure 4.8, 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 as well.

Going beyond basic partial prompt completion, I decided to try seeing if in-context learning and prompt engineering make a significant impact on credit card extraction. To begin, I added the first 3 training examples verbatim as a prefix to each prompt during evaluation. This prefix did not have an impact on how much the evaluation loop was able to extract. I then tried appending “That is not correct, try again. [User]'s credit card number is” to each incorrect generation with the hope the model would correct itself with successive attempts. This did not improve upon the baseline QLoRA either. This may have been since I was using greedy search, which is deterministic.

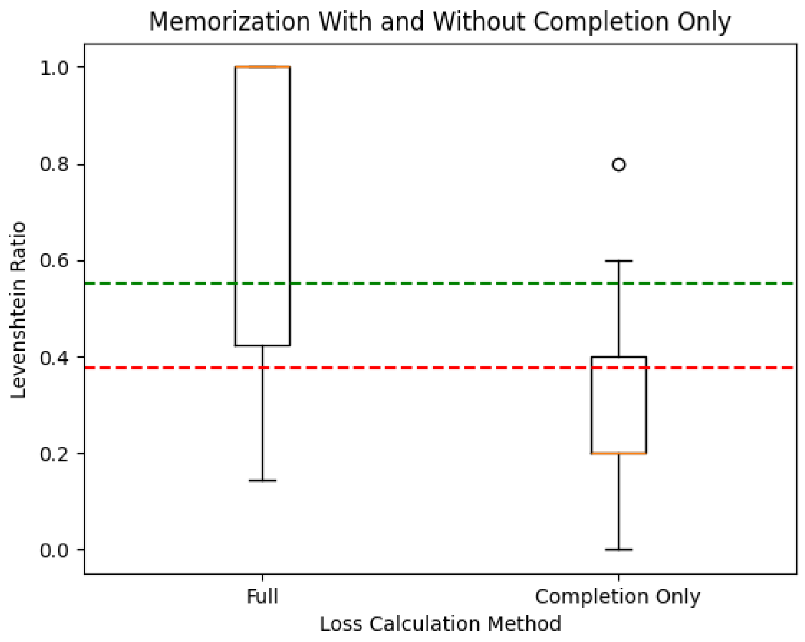


Figure 4.9: QLoRA Memorization with and without completion only loss. The model trained with completion only loss mostly memorizes worse than random guessing.

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 is only trained to minimize loss over the assistant's completion. To investigate this, I conducted a series of experiments where the model was trained with completion only loss. For comparison, trying to use partial prompt completion as with a full loss calculation fails at retrieving even a single credit card number, with the median similarity falling below that of random guessing. This comparison can be seen in Figure 4.9 where both models were trained on 100 short prompt examples, except one used completion only loss. The key takeaway from this experiment is that the partial prompt completion extraction technique is not effective in this case, performing even worse than random guessing on average.

I hypothesized that by using the loss comparison extraction technique I could still extract some information from the model. I expected the loss for the correct credit card number to be lower than the loss for incorrect credit card numbers. I thought this because the model has 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 I had hoped. Through initial experiments I found that the correct credit card numbers did have the lowest loss for some proportion of the generated completions, but also had the highest loss for a significant proportion of completions. As can be seen in Figure 4.10, the distribution of

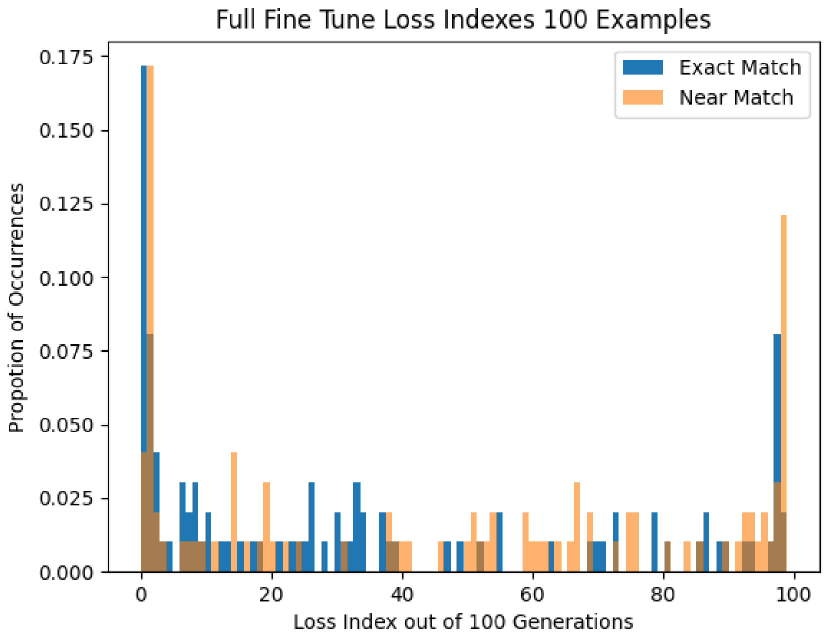


Figure 4.10: Loss comparison extraction results with model trained on 100 medium examples. The correct credit card number has a higher probability of having lowest or highest loss compared to random credit card numbers.

correct indices is bimodal. Around 25% of the correct generations have the lowest or second lowest loss compared to that of 100 random credit card numbers and around 10% of the correct generations have the highest or second highest losses. For the purposes of this analysis I will consider credit card numbers corresponding to the the two lowest and two highest loss scores as ”low” and ”high” loss credit card numbers. If a credit card number has the lowest loss score amongst 1 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, I generated 1 completion for a very similar number (the correct number with digits 5-9 replaced with ”0000”). I found 17 of the 25 correct low loss examples had a loss index of 0 and 21 of them had 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 loss index 99 (8 correct numbers had index 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 had much lower or higher loss than the others. It is important to attempt to calculate this probability to understand the risk of data extraction using this technique.

I will assume the attacker is using the fully fine-tuned model with 100 examples (the same one used in Figure 4.10). Based on the empirical data from testing, we know $P(\text{low loss}|\text{correct CC}) = 0.25$. In the first attack, the attacker keeps guessing random 16 digit credit card numbers, so $P(\text{correct CC}) = 10^{-16}$. The probability of being index 0 or 1 in a list of length 100 is 1/50, so $P(\text{low loss}) = 0.02$. We want the probability of a card number being correct given it has low loss. Using Baye's theorem, we can calculate

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

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 are miniscule probabilities, meaning that the attacker is unlikely to be able to confirm a credit card number using this technique. The main issue is that correctly picking a 16 digit credit card number randomly 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. I will assume naively that the credit card numbers in the database are uniformly distributed so that similar credit card numbers do not negatively impact loss scores for each other. Now, $P(\text{correct CC}) = 0.01$, so

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

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 low loss compared to the others, there is a 12.5% probability that it is the correct number. If it has the lowest loss, it will be a 17% probability. For high loss, they will have 5% probability. 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 made the model slightly more uniform as seen in Figure 4.11. The attack seemed to work just as well with QLoRA fine-tuning.

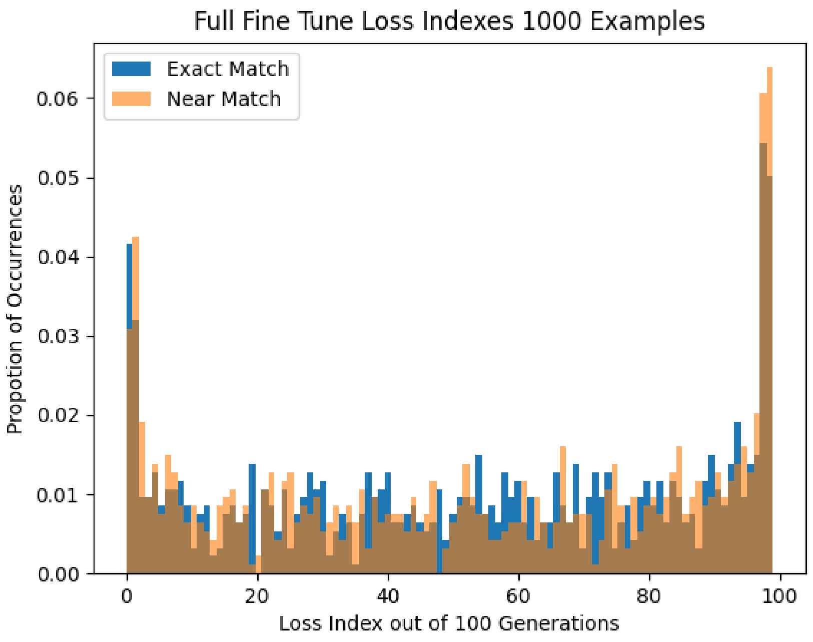


Figure 4.11: Loss comparison extraction results with model trained on 1000 medium examples. The correct credit card number has a higher probability of having lowest or highest loss compared to random credit card numbers. The probabilities are much more uniform compared to the 100 example model.

Chapter 5

Conclusion

In this work, I have shown that it is possible to extract sensitive information from a fine-tuned language model. I have shown that partial prompt completion can discover this sensitive data. I have also shown that the model can act as an oracle to confirm a credit card number by comparing loss scores. Furthermore, credit card numbers are memorized even when the model is trained with completion only loss, but that the extraction technique is not as effective.

Partial prompt completion can extract high proportions of credit card numbers in the setting where an attacker has full access to the model inputs and it has been trained for many steps. Loss comparison should be explored as a method to find out more about training data even when the model is not trained to imitate user input.

My results also confirm that trends observed in prior research hold true for fine-tuning as well. Training for more steps, with higher LoRA α, and less quantization all led to more discoverable memorization. Full fine-tuning led to nearly complete memorization at 16 epochs of training while the QLoRA memorization was more gradual and dependent more on the prompt length.

Overall, these results show how discoverable memorization can occur while fine-tuning large language models, which is a starting point to understand how to prevent data from being extracted from such models.

Further research would be needed to find a more detailed explanation for the differences in memorization between full fine-tuning and QLoRA. Possible future work could see if the model can be used to extract other types of sensitive information or key value pairs of data. Further work could also use different base models which are newer (such as Llama 3) or use different model sizes. Other extraction techniques could be explored as well, such as using in-context learning with beam search or sampling, which are more non-deterministic.

Bibliography

[1] Zeyuan Allen-Zhu and Yuanzhi Li. Physics of language models: Part 3.3, knowledge capacity scaling laws, 2024. 2.2, 4.2, 4.2

[2] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020. 1.1

[3] Nicholas Carlini, Florian Tramèr, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Úlfar Erlingsson, Alina Oprea, and Colin Raffel. Extracting training data from large language models. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2633–2650. USENIX Association, August 2021. ISBN 978-1-939133-24-3. URL `https://www.usenix.org/conference/usenixsecurity21/presentation/carlini-extracting`. 1.1, 2.1

[4] Nicholas Carlini, Daphne Ippolito, Matthew Jagielski, Katherine Lee, Florian Tramer, and Chiyuan Zhang. Quantifying memorization across neural language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL `https://openreview.net/forum?id=TatRHT_1cK`. 2.1, 4.2

[5] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms, 2023. 1.1

[6] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021. 1.1

[7] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b, 2023. 3.2

[8] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation, 2021. 1.1

[9] Yao Lu, Max Bartolo, Alastair Moore, Sebastian Riedel, and Pontus Stenetorp. Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensi-

tivity, 2022. 1.1

[10] Marius Mosbach, Tiago Pimentel, Shauli Ravfogel, Dietrich Klakow, and Yanai Elazar. Few-shot fine-tuning vs. in-context learning: A fair comparison and evaluation, 2023. 1.1

[11] Milad Nasr, Nicholas Carlini, Jonathan Hayase, Matthew Jagielski, A. Feder Cooper, Daphne Ippolito, Christopher A. Choquette-Choo, Eric Wallace, Florian Tramèr, and Katherine Lee. Scalable extraction of training data from (production) language models, 2023. 1.1, 2.1, 3.4.1

[12] Chengwei Qin, Aston Zhang, Anirudh Dagar, and Wenming Ye. In-context learning with iterative demonstration selection, 2023. 1.1

[13] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019. 1.1

[14] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models, 2020. 3.3.1

[15] Lewis Tunstall, Edward Beeching, Nathan Lambert, Nazneen Rajani, Shengyi Huang, Kashif Rasul, Alexander M. Rush, and Thomas Wolf. The alignment handbook. `https://github.com/huggingface/alignment-handbook`, 2023. 3.1