Near-Duplicate Sequence Search at Scale for Large Language Model Memorization Evaluation

Zhencan Peng
Rutgers University
zp128@scarletmail.rutgers.edu

Zhizhi Wang
Rutgers University
zw393@cs.rutgers.edu

Dong Deng
Rutgers University
dong.deng@rutgers.edu

Abstract

Recent studies show that large language models (LLM) unintendedly memorize part of the training data, which brings serious privacy risks. For example, it has been shown that over 1% of tokens generated unprompted by an LLM are part of sequences in the training data. However, current studies mainly focus on the exact memorization behaviors. In this paper, we propose to evaluate how many generated texts have near-duplicates (e.g., only differ by a couple of tokens out of 100) in the training corpus. A major challenge of conducting this evaluation is the huge computation cost incurred by near-duplicate sequence searches. This is because modern LLMs are trained on larger and larger corpora with up to 1 trillion tokens. What's worse is that the number of sequences in a text is quadratic to the text length. To address this issue, we develop an efficient and scalable near-duplicate sequence search algorithm in this paper. It can find (almost) all the near-duplicate sequences of the query sequence in a large corpus with guarantees. Specifically, the algorithm generates and groups the min-hash values of all the sequences with at least t tokens (as very short near-duplicates are often irrelevant noise) in the corpus in linear time to the corpus size. We formally prove that only $2\frac{n+1}{t+1} - 1$ min-hash values are generated for a text with n tokens in expectation. Thus the index time and size are reasonable. When a query arrives, we find all the sequences sharing enough min-hash values with the query using inverted indexes and prefix filtering. Extensive experiments on a few large real-world LLM training corpora show that our near-duplicate sequence search algorithm is efficient and scalable.

ACM Reference Format:
Zhencan Peng, Zhizhi Wang, and Dong Deng. 2023. Near-Duplicate Sequence Search at Scale for Large Language Model Memorization Evaluation. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Language models learn a probability distribution over sequences of tokens (e.g., words or byte-pair encodings [27]) and predict the next token given a sequence of previous tokens [47]. The large neural language model (LLM) is a major breakthrough in natural language processing (NLP) in recent years. They significantly boost the performance of numerous downstream NLP tasks, such as machine translation [3], text summarization [55], and question answering [38]. The state-of-the-art language models are based on Transformers [61], contain millions to billions of parameters, and are trained on large-scale text corpora with billions to trillions of tokens. For example, PaLM is a Transformer-based LLM with 540 billion parameters and is pre-trained using a high-quality corpus of 780 billion tokens [16], while GPT-3 has 175 billion parameters and is pre-trained using 500 billion token corpora [12]. Other prominent LLMs are ELMo [53], BERT [23], XLNet [68], T5 [51], etc.

A few recent studies find that LLMs unintendedly memorize part of the training data [13, 14, 36, 50, 60]. For example, Lee et al. show that over 1% of tokens generated unprompted by an LLM are part of memorized sequences in the training data [39]. Moreover, the chance a training sequence generated *verbatim* by an LLM is super-linear to the number of times it appears in the training corpus [14, 36]. In the meanwhile, existing large-scale training corpora contain numerous long duplicate sequences as well as sequences that are duplicated tens of thousands times [39]. Memorization is undesired as it not only degrades model generalization [14] but also leads to unexpected privacy risks, such as membership inference attacks [13] and training data extraction attacks [15].

However, existing work mainly focuses on the exact memorization behaviors of LLMs. In this paper, we study near-duplicates, which are much more pervasive than exact duplicates in large-scale training corpora. For example, it is estimated that around 30% to 45% of web contents are near-duplicates [11, 58]. Specifically, we propose to evaluate how many texts generated by LLMs have near-duplicate sequences in the training data. For this purpose, for each text generated by the LLM, we find all its near-duplicate sequences in the training corpus (if there are any). We define two sequences are near-duplicates if their Jaccard similarity is above a given threshold.

A major challenge of conducting this evaluation is how to efficiently find the near-duplicate sequences of a query sequence in the training corpus, which entails a huge computation cost. This is because modern LLMs are trained on larger and larger corpora (up to 1 terabyte), while the number of sequences in a text is quadratic to the text length. As pointed out by recent studies, finding exact duplicates in large-scale text corpora is already difficult [14], let alone near-duplicates. To address this issue, we develop an efficient and scalable near-duplicate sequence search algorithm based on the min-hash techniques [10]. It creates a min-hash sketch [25] for every sequence in the training corpus offline and compares the query sequence's sketch with the training sequences' sketch to find the near-duplicates. We adopt the idea from a previous work [25] to aggregate the min-hash values in a text. Moreover, we extend the previous work in the following ways. First of all, we impose a length

Conference'17, July 2017, Washington, DC, USA
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

threshold t and only generate min-hash values for sequences with at least t tokens (as very short near-sequences are often irrelevant noise). We formally prove that on average our algorithm generates $2\frac{n+1}{t+1} - 1$ min-hash values for a text with n tokens in $O(n)$ time. Thus the index time and size are reasonable even for large-scale text corpora. Second, we design a novel algorithm to efficiently find all the min-hash sketches that are similar to the query sequence's sketch. In addition, the problem definitions are slightly different. The previous work finds near-duplicate sequences in two long texts, while this paper searches sequences in a collection of texts that are similar to a query sequence. Furthermore, this paper focuses on large-scale datasets that cannot fit in memory, while the previous work only considers the in-memory case. Finally, we apply our near-duplicate sequence search algorithm to evaluate the (fuzzy) memorization behavior of large language models.

In summary, we make the following contributions in this paper.

- We develop an efficient and scalable near-duplicate sequence search algorithm. We formally analyze the impact of the length threshold in our algorithm and propose an efficient query processing algorithm.
- We conduct extensive experiments using real-world large-scale text corpora to evaluate our algorithm. Experimental results show that our algorithm is efficient and scalable.
- We evaluate the (fuzzy) memorization behaviors of four GPT-2 models of various sizes using our algorithm.

The rest of the paper is organized as follows. We briefly introduce language models in Section 2. Section 3 defines the near-duplicate sequence search problem and presents our algorithm. We evaluate our algorithm in Section 4 and evaluate large language model memorization in Section 5. Section 6 reviews related work and Section 7 concludes the paper.

2 Background: Large Language Models

In general, language models learn the probability distribution of the next token given a sequence of previous tokens. For example, given two previous tokens "*hello, good*", a reasonable language model probably assigns a higher probability to the token "*morning*" than to the token "*SIGMOD*" as the next token of the two tokens.

Training. Given a text corpus, for each training example $x_1, \cdots, x_n$ (e.g., a text in the corpus), the language model is trained to minimize the loss $\mathcal{L} = -\sum_{i=1}^{n} \log p(x_i|x_1, \cdots, x_{i-1})$ where $p(x_i|x_1, \cdots, x_{i-1})$ is the learned probability of x_i as the next token to the previous tokens $x_1, \cdots, x_{i-1}$. The target for this probability is 1 for this training example. Thus the optimal solution for the model is to memorize the training sequence [15]. However, since there are a huge number of training examples in the text corpus, the trained model usually does not memorize every sequence in the training data.

Generation Strategies. Once trained, to generate a text, we only need to repeatedly pick the next token based on the learned probability distribution. The users can optionally provide the first few tokens (namely prompt) to the language model for text generation. The simplest method to pick the next token is random sampling based on the learned probability distribution [50]. A few alternatives are greedy search, beam search, top-k sampling [24], and top-p sampling. Greedy search picks the token with the highest probability as the next token. Beam search picks the batch of next tokens with the highest probability, even though the first next token may not bear the highest probability. The top-k sampling samples only from the k most probable next tokens as predicted by the language model [24], while the top-p sampling samples only from the most probable next tokens that form the p% cumulative probability.

Memorization. It has been shown large language models memorize part of their training data. The model emits the training data verbatim when fed with appropriate prompts [14], which brings serious privacy issues [36]. For example, it is found that about 1% of tokens generated unprompted by a language model are part of sequences in the training corpus [39]. In this paper, we aim to find how many texts generated by LLMs have near-duplicate sequences in the training corpus (e.g., differ by a couple of tokens out of 100 tokens). For this purpose, we need to address the near-duplicate sequence search problem. It finds all the near-duplicate sequences of a query sequence in a large-scale training corpus (up to 1 terabyte).

3 Near-Duplicate Sequence Search

3.1 Problem Definition

We first define a few notations. A corpus D contains many texts. A text T consists of a series of tokens. The total number of tokens in a text T is denoted as $|\mathrm{T}|$. $\mathrm{T}[i, j]$ is the sequence in T from its i-th token to its j-th token (included), where $1 \le i \le j \le |\mathrm{T}|$. The token can be a word, a phrase, a byte-pair encoding (BPE) [27], etc.

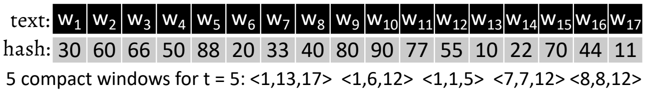
Definition 1 (Near-Duplicate Sequence Search). *Give a text corpus* D*, near-duplicate sequence search takes a query sequence Q and a similarity threshold θ as input and outputs all the sequences* $\mathrm{T}[i, j]$ *s.t.* $sim(Q, \mathrm{T}[i, j]) \ge \theta$, *where* $1 \le i \le j \le |\mathrm{T}|$ *and* $\mathrm{T} \in \mathrm{D}$.

We focus on the Jaccard similarity *sim* in this paper, which is the ratio of the intersection size (i.e., the number of common tokens) to the union size (i.e., the total number of distinct tokens) of two sequences. However, depending on how duplicate tokens are handled, there are two kinds of Jaccard similarities. The first one, *distinct Jaccard similarity*, first deduplicates two sequences and then calculates the Jaccard similarity as usual. The second one, *multi-set Jaccard similarity*, treats each occurrence of a token in a sequence as a unique token. For example, consider the two sequences (A, A, A, B, B) and (A, B, B, B, C). The distinct Jaccard similarity is 2/3, while the multi-set Jaccard similarity is 3/7 as it treats the two sequences as $(A_1, A_2, A_3, B_1, B_2)$ and $(A_1, B_1, B_2, B_3, C_1)$ and the intersection and union sizes are 3 and 7, respectively. In this paper, we use the distinct Jaccard similarity if not mentioned otherwise.

3.2 Min-Hash for Jaccard Similarity Estimation

We resort to the min-hash techniques [10] to address the near-duplicate sequence search problem. In a nutshell, given a random universal hash function[1] that maps every token to a hash value, the min-hash of a sequence is the minimum hash value of all its tokens. The distinct Jaccard similarity of two sequences can be

[1] e.g., $f(x) = a \cdot x + b \mod p$ where p is a large prime [59].



text:	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	w_{11}	w_{12}	w_{13}	w_{14}	w_{15}	w_{16}	w_{17}
hash:	30	60	66	50	88	20	33	40	80	90	77	55	10	22	70	44	11

5 compact windows for t = 5: <1,13,17> <1,6,12> <1,1,5> <7,7,12> <8,8,12>

Figure 1: A running example.

accurately estimated by s/k, where s is the number of min-hash collisions of the two sequences in k trials using k independent random universal hash functions. This is an unbiased estimation with low variance [44].

To address the near-duplicate sequence search problem, we develop an algorithm to find all the sequences in the corpus whose min-hash values collide with those of the query sequence at least $\lceil k\theta \rceil$ times, where θ is the user-provided similarity threshold. In addition, in practice, only near-duplicate sequences that are long enough are interesting. For this purpose, we impose a *length threshold t* and only find near-duplicate sequences with at least t tokens. Formally, we have the following problem definition.

Definition 2. *Give a text corpus* D*, a length threshold t, and k independent random universal hash functions* $f_1, \cdots, f_k$*. Near-duplicate sequence approximate search takes a query sequence Q and a threshold θ as input and outputs all the sequences* $\mathrm{T}[i,j]$ *s.t.* $\sum_{x=1}^{k} \mathbf{1}\{f_x(Q) = f_x(\mathrm{T}[i,j])\} \geq \lceil k\theta \rceil$*, where* $\mathrm{T} \in \mathrm{D}$ *and* $j - i + 1 \geq t$*.*

Note here the hash function f_x outputs the min-hash of its input sequence. In addition, $\mathbf{1}\{b\}$ is a boolean function that returns 1 (or 0) when b is true (or false). Since the variance of the Jaccard similarity estimation is $O(1/k)$ [44], for a large enough k, the near-duplicate sequence approximate search guarantees to find most of the sequences in the corpus that are similar to the query sequence.

3.3 Efficient Min-Hash Generation

To find all the near-duplicate sequences, we propose to generate k min-hash values for every sequence (of length at least t) in the text corpus during the offline indexing phase. However, the total number of sequences in a large-scale text corpus (e.g., consists of a few hundreds of billions of tokens) is enormous. A recent work Allign on finding all the near-duplicate sequences in two long texts designs an algorithm to tackle this problem [25]. In this paper, we adapt the algorithm to work with the distinct Jaccard similarity, improve its time complexity, and formally analyze the impact of the length threshold t. Finally, we design an algorithm for near-duplicate sequence approximate search based on it.

The Existing Work for Text Alignment. Allign proposes an algorithm to efficiently generate the min-hash values of all the sequences in a long text [25]. The key idea is to group nearby sequences in a text by their min-hash values using *compact windows*. A compact window is a tuple $\langle \mathrm{T}, f, l, c, r \rangle$. It represents all the sequences $\mathrm{T}[i,j]$ where $l \leq i \leq c \leq j \leq r$ and all these sequences have the same min-hash, which is $f(\mathrm{T}[c])$. Moreover, the compact window is maximal, i.e., extending either l or r makes the above condition no longer hold. Clearly, by definition, the hash value of $\mathrm{T}[c]$ is the smallest among all the tokens in $\mathrm{T}[l,r]$. For simplicity, we omit the text T and hash function f in the compact window when they are clear from the context. For example, consider the text T with 17 tokens and their hash values derived from a random hash function f as shown in Figure 1. $\langle 1, 13, 17 \rangle$ is a compact window.

```
Algorithm 1: Indexing
  Input: D: a text corpus; $k$: an integer; $f_1, f_2, \cdots, f_k$: $k$
         independent hash functions; $t$: a length threshold.
  Output: $k$ inverted index files of compact windows on disk.
  load the text corpus D into memory;
  foreach $1 \leq i \leq k$ do
      foreach text $\mathrm{T} \in \mathrm{D}$ do
          GenerateCompactWindows($1, |\mathrm{T}|, \mathrm{T}, f_i, t, Q$);
          foreach compact window $\langle l, c, r \rangle \in Q$ do
              $h \leftarrow f_i(\mathrm{T}[c])$;
              append $\langle \mathrm{T}, l, c, r \rangle$ to the inverted list $I_i[h]$;
      write the inverted index $I_i$ to the disk as a file;
  // for large-scale corpora, load one batch of
     texts at a time, partition the compact windows
     by i and h, and use hash aggregation to build
     the inverted index files for each partition.
```

```
Algorithm 2: GenerateCompactWindows($l, r, \mathrm{T}, f, t, Q$)
  Input: $l$: an integer; $r$: an integer; T: a text; $f$: a hash function;
         $t$: a threshold; $Q$: a collection of compact windows.
  if $r - l + 1 < t$ then return;
  find a position $c \in [l,r]$ s.t. $\forall p \in [l,r], f(\mathrm{T}[c]) \leq f(\mathrm{T}[p])$
    using an advanced RMQ algorithm [26], break ties arbitrarily;
  add a compact window $\langle l, c, r \rangle$ to $Q$;
  GenerateCompactWindows($l, c-1, \mathrm{T}, f, t, Q$);
  GenerateCompactWindows($c+1, r, \mathrm{T}, f, t, Q$);
```

All the sequences $\mathrm{T}[i,j]$ where $1 \leq i \leq 13 \leq j \leq 17$ share the same min-hash value $f(\mathrm{T}[13]) = 10$.

Allign proves there are $O(n)$ compact windows in a text with n tokens and these compact windows and all the sequences in the text are surjective, i.e., each sequence is in one and only one compact window. It develops an algorithm that generates all the $O(n)$ compact windows in the text in $O(n \log n)$ time and $O(n)$ space. Moreover, it extends the algorithm to deal with the multi-set Jaccard similarity when the text contains duplicate tokens [25].

Our Min-Hash Generation. As only the min-hash values of sequences with at least t tokens are needed in our settings, we do not need to generate a compact window $\langle l, c, r \rangle$ if its "width" $r - l + 1 < t$. Next we present an algorithm that generates all the "valid compact windows" whose widths are at least t in a text with n tokens in $O(n)$ time and space. We further prove our algorithm generates $2\frac{n+1}{t+1} - 1$ valid compact windows on average (i.e., in expectation) and every sequence with at least t tokens is in one and only one of these valid compact windows. The compact windows can be used to accurately estimate the distinct Jaccard similarity to the query.

Our algorithm is similar to the one in Allign. It is a divide-and-conquer algorithm. Given a sequence $\mathrm{T}[l,r]$, it divides the sequence into two by the token $\mathrm{T}[c]$ with the smallest token hash value in the sequence. Then it recursively solves two sub-problems, one takes the (sub)-sequence $\mathrm{T}[l, c-1]$ as the input and the other one takes $\mathrm{T}[c+1, r]$ as the input. In addition, it produces a tuple $\langle l, c, r \rangle$, which, by definition, must be a compact window. The recursion stops when the input sequence is not long enough (more specifically, when

$r - l + 1 < t$) as no valid compact window exists in the input. Note initially the input sequence is the entire text T[1, |T|].

Note that, when there are multiple tokens with the same smallest hash value in the input sequence (this happens when the text contains duplicate tokens), we randomly choose one to divide the input sequence (i.e., break ties arbitrarily). The pseudo-code of the divide-and-conquer algorithm is shown in Algorithm 2. It takes a sequence $\mathrm{T}[l, r]$, a length threshold t, a hash function, and a result set Q as its input. If the input sequence is too short, the recursion stops (Line 1); otherwise, it finds a token $\mathrm{T}[c]$ in the input sequence with the smallest hash value, adds a compact window $\langle l, c, r\rangle$ into the result set Q (Lines 2 to 3), and recursively generates the compact windows in the two (sub)-sequences divided by $\mathrm{T}[c]$ (Lines 4 to 5).

Example 1. For example, consider the text and its token hash values in Figure 1. Let the length threshold be $t = 5$. The algorithm first chooses T[13] to divide the text T[1, 17] to two sequences T[1, 12] and T[14, 17] and generates a compact window $\langle 1, 13, 17\rangle$. The second sequence is shorter than t and is skipped. The algorithm recursively divides the first sequence by T[6] to two sequences T[1, 5] and T[7, 12] and generates a compact window $\langle 1, 6, 12\rangle$. Eventually, it generates 5 "valid" compact windows that are wide enough. The number exactly matches the expectation (as described presently), which is $2\frac{n+1}{t+1} - 1 = 2\frac{18}{6} - 1 = 5$.

Theorem 1. *Algorithm 2 generates* $2\frac{n+1}{t+1} - 1$ *compact windows for a text* T *with* n *distinct tokens in expectation. Furthermore, every sequence in* T *with at least* t *tokens is in one and only one of the generated compact windows.*

Proof. Let S_n denote the expected number of compact windows generated by the algorithm for a sequence of length n. Since the token hash values are random, every distinct token in the input sequence has the same probability $\frac{1}{n}$ to be the token that divides the input sequence. Thus we have

$$S_n = \sum_{i=1}^{n} \frac{1}{n}(S_{i-1} + 1 + S_{n-i}) = 1 + \frac{2}{n}\sum_{i=1}^{n} S_{i-1}.$$

The base cases are $S_0 = S_1 = \cdots = S_{t-1} = 0$ and $S_t = 1$. Solving the recursive formula, we have $S_n = 2\frac{n+1}{t+1} - 1$.

The second part of the lemma can be proved using reduction. Let c be the first token where the algorithm chooses to divide the text. The algorithm must generate a compact window $\langle 1, c, n\rangle$. All the sequences of T can be partitioned into three categories, $\mathrm{T}[i_1, j_1]$ where $1 \le i_1 \le j_1 < c$, $\mathrm{T}[i_2, j_2]$ where $1 \le i_2 \le c \le j_2 \le n$, and $\mathrm{T}[i_3, j_3]$ where $c < i_3 \le j_3 \le n$. Sequences in the second category must be represented by the generated compact window $\langle 1, c, n\rangle$ once and only once. Moreover, based on the reduction, all the sequences in the first (or third) category must be in one and only one compact windows generated by the algorithm when the input is $\mathrm{T}[1, c-1]$ (or $\mathrm{T}[c+1, n]$). The base case is when the input sequence is shorter than t, in which case, all its sub-sequences are shorter than t and no compact window is needed to be generated. □

Complexity Analysis. Allign uses a segment tree to find a token with the smallest hash value in the input sequence (which is a classical range minimum query, RMQ), which takes $O(\log n)$ for each of the $O(n)$ recursions. However, more advanced RMQ data structures and algorithms are available [2, 8, 26]. For example, the data structure designed in [26] can be constructed in $O(n)$ time and space and it answers an RMQ in O(1) time. Thus the time and space complexities of our compact window generation algorithm can be reduced to $O(n)$ using this data structure [26].

3.4 Indexing Compact Windows

In this section, we discuss how to index the generated compact windows. We propose to build k inverted index files. In each inverted index I_i, the compact windows $\langle \mathrm{T}, f_i, l, c, r\rangle$ sharing the same min-hash $h = f_i(\mathrm{T}[c])$ are placed in the same inverted list $I_i[h]$ ordered by the text identifiers T. When a query sequence arrives, we first get its k min-hash values, then retrieve the k corresponding inverted lists from the k inverted indexes, and finally count the hash collisions to determine the near-duplicate sequences.

We first consider medium-scale corpora such as OpenWebText [30] (around 31 GB after tokenization) that can fit in memory. Note we target at a single ordinary machine with around 64 GB memory and length threshold $t \ge 25$. We assume each token is an integer and the number of texts fits in a 4-byte integer. As shown in Algorithm 1, we first load the entire corpus in memory (Line 1). For each of the k hash functions, the algorithm first builds an inverted index in memory and then writes it back to disk (Lines 2 to 8). This is feasible as each inverted index contains no more than $\frac{2N}{t+1}$ compact windows on average, where N is the total number of tokens in the corpus (i.e., the dataset size, which is ~31 GB for OpenWebText). Since each compact window $\langle \mathrm{T}, l, c, r\rangle$ consists of 4 integers (note the hash function is the same for all the compact windows in the same inverted index and can be ignored). The ratio of the index size to the corpus size is no more than $\frac{8}{t+1}$ on average. Thus the size of each inverted index is much smaller than the medium-scale corpus for a reasonable length threshold t (e.g., 50). For large-scale corpora like C4 [50] (around 750 GB after tokenization) and Pile [28] (around 825 GB) that cannot fit in memory, we use hash aggregation [52, 57] to build the inverted index files. Specifically, we load a batch of texts at a time and generate their compact windows. For each of the k hash functions, we partition the generated compact windows such that compact windows from the same i-th hash function and with the same min-hash value h are in the same partition. Finally, we load each partition into memory to build the inverted list $I_i[h]$ and write them back to disk to construct the inverted index. In case a partition cannot fit in memory, we use recursive partitioning [52]. The hash aggregation entails two passes of the inverted indexes (one read and one write).

We can also build the index in parallel. Specifically, we assign each thread a batch of texts and a private memory space. Each thread generates compact windows for all its texts and writes the compact windows to its private memory. Finally, the compact windows in the private memory space are merged and flushed to disk.

3.5 Query Processing

Once a query sequence Q arrives, we first calculate its k min-hash values (a.k.a., k-mins sketch [25]) and load the k corresponding inverted lists into memory. Each of them contains a list of compact windows $\langle \mathrm{T}, l, c, r\rangle$, in which every sequence $\mathrm{T}[i, j]$ where $l \le i \le$

Algorithm 3: NearDuplicateSearch

```
Input: $Q$: a query sequence; $\theta$: a similarity threshold;
       $f_1, f_2, \cdots, f_k$: $k$ independent hash functions.
Output: All the near-duplicate sequences of $Q$ in the corpus.
begin
    get the $k$ min-hash values of $Q$ using $f_1, \cdots, f_k$;
    load the short inverted lists $I_1, I_2, \cdots, I_p$ into memory;
    group the compact windows by their texts;
    foreach group C of text T of size $\geq \beta - (k - p)$ do
        $A$ = CollisionCount(C, $\beta - (k - p)$);
        if $A$ is not empty then
            locate and load the compact windows of T in the
            $k - p$ long inverted lists and add them to C;
            $A'$ = CollisionCount(C, $\beta$);
            foreach ([$x, x'$], [$y, y'$]) in $A'$ do
                foreach $i \in [x, x']$ and $j \in [y, y']$ do
                    add near-duplicate sequence T[$i, j$] to $\mathcal{Q}$;
    return $\mathcal{Q}$;
end
```

Algorithm 4: CollisionCount(C, α)

```
Input: C: a collection of compact windows from the same text;
       $\alpha$: a collision threshold.
Output: Interval pairs containing all the sequences contained
        by at least $\geq \alpha$ compact windows.
$\mathcal{X} \leftarrow$ the left intervals $[l, c]$ of all compact windows in C;
$A$ = IntervalScan($\mathcal{X}, \alpha$);
foreach (C$'$, [$x, x'$]) in $A$ do
    $\mathcal{Y} \leftarrow$ the intervals $[c, r]$ of all compact windows in C$'$;
    $A'$ = IntervalScan($\mathcal{Y}, \alpha$);
    foreach (C$''$, [$y, y'$]) in $A'$ do
        add the pair ([$x, x'$], [$y, y'$]) to the result set $A''$;
return $A''$;
```

Algorithm 5: IntervalScan($\mathcal{X}, \alpha$)

```
   Input: $\mathcal{X}$: a collection of intervals; $\alpha$: a collision threshold.
   Output: All subsets of $\mathcal{X}$ with non-empty overlap and size $\geq \alpha$.
1  foreach interval ($W$, [$x, y$]) in $\mathcal{X}$ do
2      add endpoints ($x$, 1, $W$) and ($y + 1$, 0, $W$) into ep;
3  sort the endpoints in ep in ascending order;
4  foreach distinct endpoint e in ep do
5      foreach endpoint ($e, b, W$) in ep do
6          if b is 1 then add $W$ into C;
7          if b is 0 then remove $W$ from C;
8      if $|C| \geq \alpha$ then
9          add (C, [$e$, next distinct endpoint)) to $A$;
10 return $A$;
```

$c \leq j \leq r$ collides once with the query sequence. To find all the near-duplicate sequences in a text that collide with the query sequence enough times (i.e., at least $\lceil k\theta \rceil$ times), we aggregate the compact windows in the k inverted lists by their text identifiers T. For each group of compact windows, we aim to find all the sequences T[i, j] *reside in* at least $\beta = \lceil k\theta \rceil$ compact windows $\langle l, c, r \rangle$ in the group, i.e., $l \leq i \leq c \leq j \leq r$.

For this purpose, we split each compact window $\langle l, c, r \rangle$ into two parts, the left interval $[l, c]$ and the right interval $[c, r]$. For any subset of compact windows in the group, let $[x, x']$ be the overlap of their left intervals and $[y, y']$ be the overlap of their right intervals. Then, every sequence T[i, j] where $i \in [x, x']$ and $j \in [y, y']$ must collides with the query sequence s times, where s is the number of compact windows in the subset. If $s \geq \beta = \lceil k\theta \rceil$, T[$i, j$] must be a near-duplicate sequence of the query sequence.

Based on the above observation, we propose an algorithm CollisionCount to find all the "large enough" subsets of compact windows whose left intervals and right intervals both have non-empty overlaps. It processes the left intervals and right intervals separately using our IntervalScan method. In a nutshell, given a collection of intervals, IntervalScan first collects the endpoints of all intervals. Then, it sorts the endpoints in ascending order and visits them one by one. As each endpoint either means the start (entrance) of an interval or the end (exit) of an interval, we can keep track of the subset of intervals that already start but not end yet during visiting. Clearly, the overlap of the subset of intervals must be non-empty. Thus we report the subset if its size is "large enough".

The pseudo-code of IntervalScan is shown in Algorithm 5. It takes a collection of intervals $\mathcal{X}$ and an integer threshold α as input and reports all the subsets of $\mathcal{X}$ whose overlaps are non-empty and whose sizes are at least α. The algorithm first collects the two endpoints x (means the interval starts) and $y + 1$ (means the interval exits) of every interval $[x, y]$ in the input $\mathcal{X}$ (Lines 1 to 2). Then, it sorts all the endpoints in ascending order and visits them in sequence (Lines 3 to 5). For each starting endpoint, its corresponding interval is added to an array C (Line 6). For each ending endpoint, its corresponding interval is removed from the array C (Line 7). Once a distinct endpoint x is passed, we check the status of the array. Let the next distinct endpoint be x'. Then, $[x, x')$ must be part of the overlap of all the intervals in the array right now. This is because these intervals all have started but not end yet in $[x, x')$. If there are at least α intervals in the array, we report it, as well as the part of their overlap $[x, x')$ (Lines 8 to 9).

Lemma 1. *IntervalScan generates every subset of $\mathcal{X}$ whose overlap is non-empty and whose size is at least α once and only once.*

We omit the proof due to space limit. Based on the IntervalScan method, we can find all the "large enough" subsets of compact windows whose left and right intervals both have non-empty overlaps. As shown in Algorithm 4, it takes a group of compact windows from the same text T and an integer threshold as input. It first collects the left intervals $[l, c]$ of every compact window $\langle l, c, r \rangle$ in the group. Then, it finds all the subsets of "large enough" left intervals with non-empty overlaps using IntervalScan. For each of the subset, it collects the right intervals of the compact windows from where the left intervals in the subset come. It uses IntervalScan again to find those large enough subset of right intervals with non-empty overlaps. Finally, it adds the pair of non-empty interval overlaps to the result set and returns the result set finally.

Complexity Analysis. Suppose there are m compact windows in the group. The time complexity is $O(m^2 \log m)$. This is because it generates at most $O(m)$ large enough subsets of left intervals with non-empty overlaps. For each of them, it takes $O(m \log m)$ to sort the endpoints of the right intervals. The scan takes linear time to m. Thus the total time complexity is $O(m^2 \log m)$. Note the size of each compact window group is usually small. In addition, the I/O cost dominates the query latency. Thus the time complexity of our algorithm is affordable.

Prefix Filtering to Avoid Long Inverted Lists. Although each (distinct) token has the same chance to be the min-hash of a sequence, the lengths of their inverted lists are vastly different. This is because, in our compact window generation algorithm (that designed specifically for the distinct Jaccard similarity instead of the multi-set Jaccard similarity), if a token has the minimum hash value in the input sequence, each occurrence of the token in the sequence may produce a compact window, which is placed in the same inverted list. Thus the length of the inverted list is proportional to the token frequency. In the meanwhile, it is well known that the word/token frequency in natural languages follows the Zipf law [49], i.e., the frequency of the most frequent token is twice that of the second most frequent token, three times that of the third most frequent token, etc. Thus in each inverted index, there are a few very long inverted lists.

When a query sequence contains min-hash values with long inverted lists, it is time consuming to read the entire inverted lists. To avoid this, we use the prefix filtering techniques [6, 7, 19, 65]. Specifically, among the k inverted lists, we only load those whose lengths are smaller than a threshold. Suppose there are p of them. Then we use our CollisionCount algorithm to find all the candidates that collide at least $\beta-(k-p)$ times. For each text T in the candidates, we locate its compact windows in the rest long inverted lists and only load their compact windows into memory. After that, we re-apply our CollisionCount algorithm to produce the final near-duplicate sequences. The pseudo-code is shown in Algorithm 3.

Zone Map. To facilitate locating compact windows of a specific text in an inverted list, we create a zone map [52] for the long inverted lists. Specifically, since the compact windows are ordered by the text identifiers in the inverted list, we record the offset of every other s text identifiers in the inverted list, where the step size s is a parameter. A few works design cost-models to choose a good cutoff of long and short inverted lists (a.k.a., prefix length) [7, 22, 62]).

Theorem 2. *Algorithm 3 is sound and complete. The sequences generated by the algorithm are all (approximate) near-duplicate to the query sequence and all (approximate) near-duplicate sequences of the query sequence are generated by the algorithm.*

Remark. In practice, it is undesired to enumerate and show all the (redundant) near-duplicate sequences to the users. Instead, we merge the overlapping near-duplicate sequences such that all the sequences we report are disjoint to each other.

4 Evaluating Near Duplicate Sequence Search

Datasets. We used two real-world datasets. Both of them are frequently used in large neural language model pre-training. (1) OpenWebText is a collection of web texts highly ranked on Reddit [30]. It is an open-source replication of the WebText dataset, which is used to train the LLM GPT-2 [50]. Note that both exact and near-duplicate texts in OpenWebText have been removed. We downloaded the dataset from huggingface[2]. It consists of around 8 million texts and the raw size is around 40 GB. (2) The Pile [28] is constructed from 22 diverse high-quality datasets. We downloaded it from huggingface[3]. Its raw size is 825.18 GB. It was used to trained the LLM GPT-Neo[4].

BPE Tokenization. For OpenWebText, we trained a BPE model with vocabulary size of 64000 using 1 million texts with maximum length 10,000. After tokenization using the BPE model, the sizes of OpenWebText was respectively 31GB (note that we used a 4-byte integer to represent a token). For Pile, we used the GPT2Tokenizer[5] to tokenize the dataset. This BPE tokenizer's vocabulary size is 50257. The dataset size after tokenization was 649 GB.

Environment. We implemented our algorithm using C++ and compiled the programs using g++7.5 with -O3 optimization. All the experiments were conducted on a machine with 24 2.40GHz Intel Xeon Gold 6212U CPU cores (48 threads with hyper-threading) and 64 GB memory and 20 TB hard disk. The operating system is Ubuntu 18.04. We used OpenMP for parallel computation.

4.1 Evaluating Index Construction

In this section, we evaluate our compact window generation and indexing algorithms.

Number of Compact Windows Generated: We first evaluate the number of compact windows generated under various length thresholds t, numbers of hash functions k, vocabulary sizes, and dataset sizes n. As shown in Figures 2(a)-2(b), when we increased the length threshold t, the numbers of compact windows generated linearly decreased. For example, for $t = 25$, 50, and 100, the numbers of compact windows generated were around 620 millions, 330 millions, and 180 millions for $k = 1$, 32K vocabulary size, and 8 million OpenWebText texts. This is because the number of compact windows generated in expectation is $2\frac{n+1}{t+1} - 1$, which is inversely proportionally to the length threshold t. In addition, for the same length threshold, a larger vocabulary size resulted in a bit less compact windows. This is because the number of tokens n in a text after encoding using a larger vocabulary was usually a little smaller, while the number of compact windows is proportional to n. Furthermore, the number of compact windows generated grew linearly with the number of hash functions k. Moreover, as shown in Figures 2(c)-2(d), when we increased the corpus size, the number of compact windows generated grew linearly. For example, for 1M (million), 2M, 4M, and 8M OpenWebText texts, with fixed $k = 1$, vocabulary size 64K, and $t = 100$, the numbers of compact windows generated were respectively 23 millions, 46 millions, 92 millions, and 183 millions. This is consistent with our theoretical analysis.

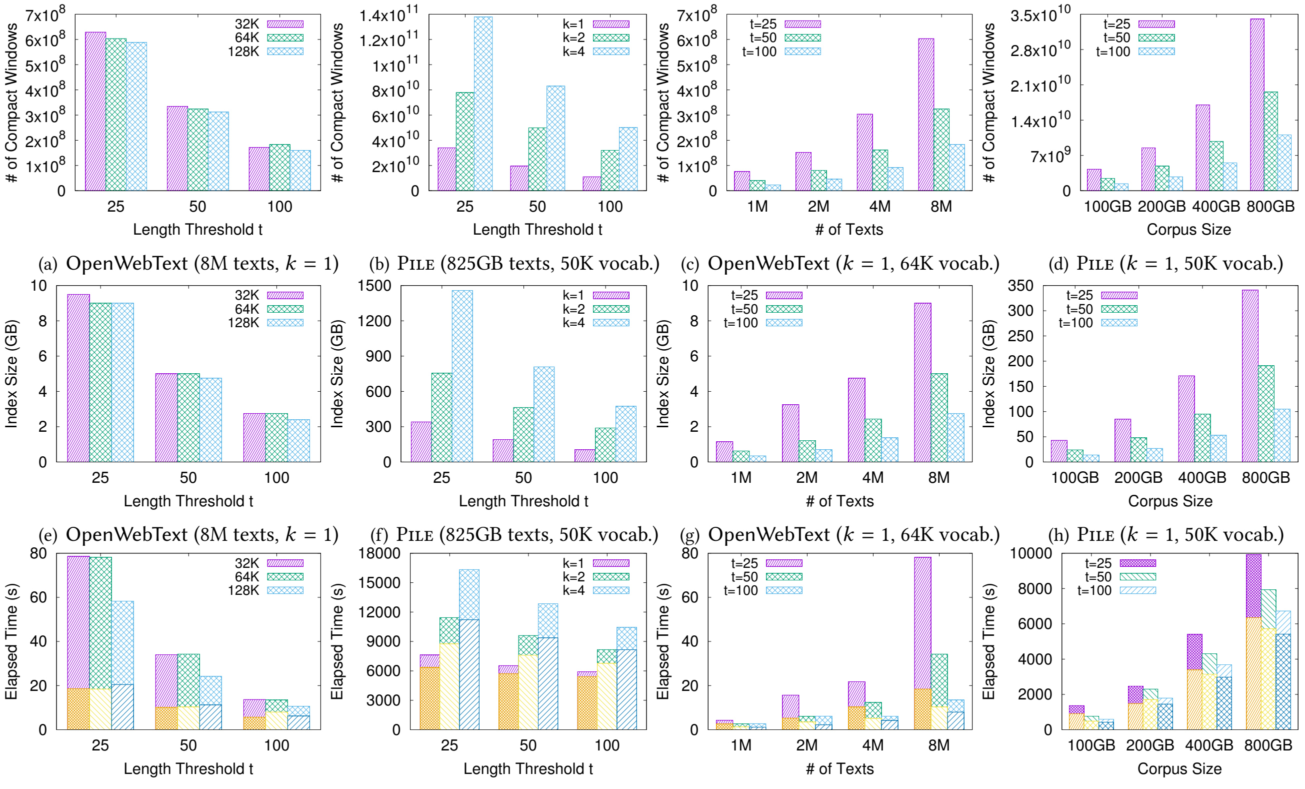
Index Size. Next we evaluate the index sizes. Figures 2(e)-2(h) show the results. The index size was proportional to the number of compact windows and showed the same trends as the number of compact windows. As we can see, each inverted index was only around 2 GB when $t = 100$ on OpenWebText, while the dataset size

[2] https://huggingface.co/datasets/openwebtext

[3] https://huggingface.co/datasets/the_pile

[4] https://huggingface.co/docs/transformers/model_doc/gpt_neo

[5] https://huggingface.co/docs/transformers/model_doc/gpt2#transformers.GPT2Tokenizer



(a) OpenWebText (8M texts, $k = 1$) (b) Pile (825GB texts, 50K vocab.) (c) OpenWebText ($k = 1$, 64K vocab.) (d) Pile ($k = 1$, 50K vocab.)

(e) OpenWebText (8M texts, $k = 1$) (f) Pile (825GB texts, 50K vocab.) (g) OpenWebText ($k = 1$, 64K vocab.) (h) Pile ($k = 1$, 50K vocab.)

(i) OpenWebText (8M texts, $k = 1$) (j) Pile (825GB texts, 50K vocab.) (k) OpenWebText ($k = 1$, 64K vocab.) (l) Pile ($k = 1$, 50K vocab.)

Figure 2: Evaluating Index Construction.

after tokenization was around 31 GB. For Pile, each inverted index was around 100 GB when $t = 100$, while the raw dataset size was 825 GB. Although k inverted indexes were constructed in total, the index size was reasonable compared to the dataset size.

Index Time. We report the index time in Figures 2(i)-2(l). The index time consists of the compact window generation time (the lower bars in the figures) and the disk I/O cost (the upper bars in the figures). As we can see, the index time was also linear to the dataset size and the number of hash functions, while inversely linear to the length threshold.

4.2 Evaluating Query Processing

In this section, we evaluate our query processing algorithm. We downloaded a collection of texts generated by GPT-2 released by OpenAI (the creator of GPT-2) [6] and randomly chose a few texts as the query sequences for OpenWebText. For Pile, we first generated a few texts using the GPT-Neo-1.3B model without prompt. Then we slide a fixed-width window of 64 tokens over the generated texts as the query sequences. We first vary the number of hash functions k and the similarity threshold θ and report the query latency and the number of near-duplicates found. Note the query latency consists of two parts, the IO cost for loading inverted indexes (lower bars in the figures) and the CPU computation cost (upper bars in the figures). In addition, all the experimental results were averaged over 100 random queries. Figures 3(a), 3(b), 3(e), and 3(f) show the results. As we can see from the figures, when the similarity threshold decreased, the query latency significantly increased. Furthermore, query latency was dominated by the IO cost when the similarity threshold was low. This is because prefix filtering did not filter all the sequences. A few texts need to access their zone maps and long inverted lists, which incurred significant IO cost. There was no clear trend between the number of hash functions and the query latency. This is because for different k, the filtering power of prefix filter differs. Furthermore, no exact duplicates (i.e., when the similarity threshold $\theta = 1$) were found for the 100 random query sequences, while for $\theta = 0.7$, on average 13 near-duplicate sequences were found in OpenWebText.

Next, we vary the dataset size, the similarity threshold, and the length threshold and report the query latency. Figures 3(c), 3(g) and 3(h) show the results. As we can see, when the dataset size increased (i.e., the number of texts in the corpus), the query latency linearly increased. This is because the inverted index grows linearly with the dataset size, while both the IO cost and the computation cost grow linearly with the dataset size. Moreover, for large dataset sizes, the IO cost dominated the query latency. Furthermore, the query latency was inversely proportional to the length threshold. This is because the large length threshold results in less number of compact windows and shorter inverted lists. Figure 3(d) shows the query latency under various prefix lengths from 5% most frequent tokens to 20% most frequent ones. We can see the total query latency stayed roughly the same. However, the IO cost was proportional to

[6] https://github.com/openai/gpt-2-output-dataset

the prefix length, while the CPU computation cost was inversely proportional.

5 Evaluating Language Model Memorization

Settings. We focus on the GPT-2 [50] language models, which are Transformer-based neural language model. Specifically, we downloaded the Mistral family pre-trained GPT-2 models[7]. It contains 5 small (117M parameters) and 5 medium (345M parameters) GPT-2 models. These models were trained using the OpenWebText dataset. For each model, it has many checkpoints of the model in different training steps. In our experiments, we used the small and the medium GPT-2 models with seed 21 at training step 400,000. Furthermore, we downloaded two GPT-Neo language models[8]. The GPT-Neo-1.3B model contains 1.3 billion parameters, while the GPT-Neo-2.7B model has 2.7 billion parameters. These models were trained using the Pile dataset. For each of the four language model, similar to the previous work [39], we used the top-50 sampling [38] strategy to generate texts without prompts. The lengths of the generated texts were up to 512 tokens. The first column in Table 1 shows a couple of example texts (snippets) generated by GPT-Neo-2.7B.

Evaluating Memorization. To evaluate the memorization behaviors in a reasonable time, given a text T generated by the models, we used all the fixed-length sequences $\mathrm{T}[i \cdot x + 1, (i + 1) \cdot x]$ in the text as the query sequences where x is the fixed query sequence length and $(i + 1) \cdot x \leq |\mathrm{T}|$. Then we used our near-duplicate sequence search algorithm to find near-duplicate sequences of the query sequences in the training corpus. Finally, we report the ratio of query sequences having near-duplicates in the training corpus over all the evaluated query sequences. Table 1 lists a couple of sequences generated by GPT-Neo-2.7B and their near-duplicate sequences we found in the training dataset Pile.

We first evaluate the memorization behaviors of language models of various sizes. We set $x = 32$, $t = 25$, and $k = 32$ and varied the similarity threshold θ. Figures 4(a) and 4(c) show the results. As we can see, with the decrease of the similarity threshold, the percentage of generated texts having near-duplicates in the training corpus increased. For example, there were around 2.3%, 3.3%, and 4.8% of sequences generated by GPT-Neo-1.3B having near-duplicate sequences in the training corpus Pile when the similarity threshold were 1.0, 0.9, and 0.8. Furthermore, the GPT-Neo-2.7B model memorized more sequences than the GPT-Neo-1.3B model. For example, when $\theta = 0.8$, around 7.2% and 4.8% of sequences generated by GPT-Neo-2.7B and GPT-Neo-1.3B were memorized respectively. This is consistent with previous studies [39], which find that language models with more parameters tend to memorize more training data. However, the small model with 117M parameters in the Mistral GPT-2 family memorized more sequences than the medium model with 345M parameters. It may because the model sizes were not large enough. Note the previous work [39] used a language model with 1.5 billion parameters.

We also measured the impact of the sliding window width x (i.e., the query sequence length). Figures 4(b) and 4(d) show the results. As we can see, the smaller sliding window usually entailed a greater percentage of memorized sequences. This is because short sequences are more likely to have near-duplicate sequences. The reason that sliding window width $x = 128$ memorized more percentage of generated sequences than $x = 64$ for the GPT-Neo-2.7B model was because the number of sliding windows (i.e., query sequences) of width 64 is more than twice the number of sliding windows of width 128 (as the last 64-token sliding window in a text may not be in the last 128-token sliding window in the text).

6 Related Work

Near-Duplicate Search and Detection. Near-duplicate detection has been extensively studied in many fields [1, 17, 18, 48, 54, 63, 66, 67]. There are various definitions of near-duplicates based on the data model (using q-grams, tokens, or characters as the units), the metrics (weighted and unweighted, Jaccard similarity [40], cosine similarity, overlap similarity, edit distance, Soundex distance, etc), and the problem settings (similarity joins [20, 42], similarity search [19], approximate extraction [41], approximate alignment [25], etc). A frequently used heuristic for near-duplicate search is seed-and-extend [5, 9, 11, 32, 34, 37, 45, 48, 54, 56, 63]. It first finds seed matches between the query sequence and the data sequence and then extends the seed matches as far as possible. However, this heuristic does not have any guarantee. Moreover, it usually only works for order-sensitive similarity metrics. For Jaccard similarity, a sequence is a set of unordered tokens. Thus it is hard, if not impossible, to apply the heuristic. Moreover, it is suspicious if the heuristic would work for terabyte data. Two most relevant works are Allign [25] and TxtAlign [64]. TxtAlign focuses on text alignment, which takes two texts as input and finds all the near-duplicate sequence pairs in the two texts. Allign focuses on partial plagiarism detection, which detects near-duplicate sequences between a query document and every data document.

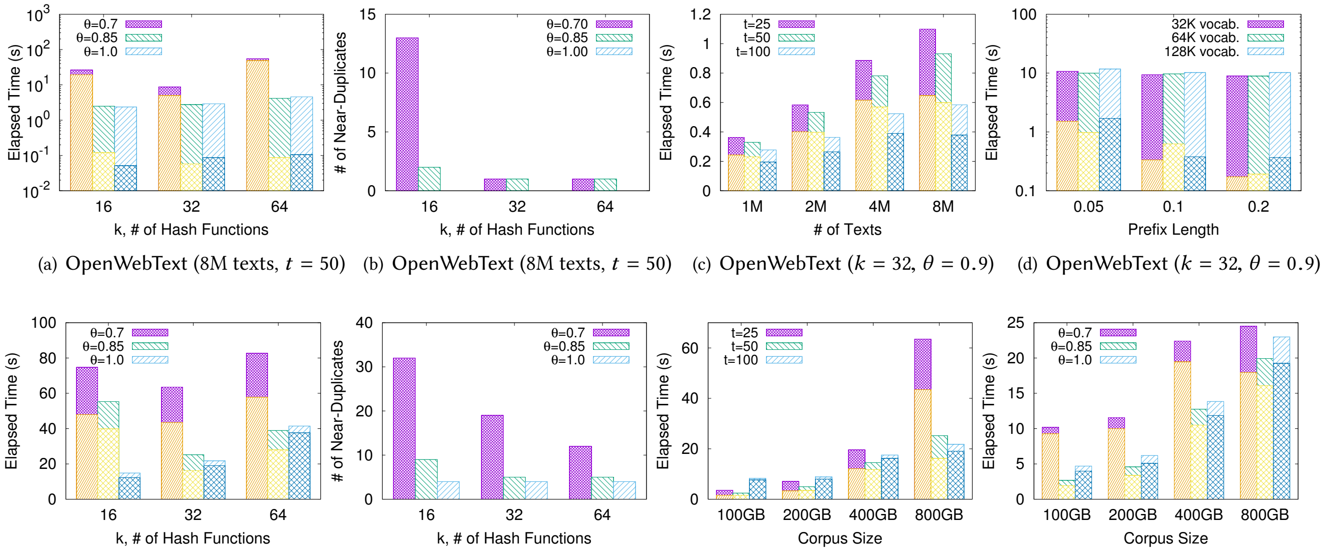
Full-Text Search and Search Engine. Full-text search and search engine support keyword searches, which finds all the documents containing the query keywords [31, 33, 35, 43]. Fuzzy match, regular expression, boolean operators, and wildcards can be used for keyword matches [4, 21]. For example, AI2 maintains a full-text search service for the C4 dataset using ElasticSearch[9] [29]. Full-text search and search engine cannot handle near-duplicate sequence search, which is much more computational intensive.

Large Language Model Memorization Evaluation. Many studies show large, neural language models memorizes part of the training data. However, existing works mostly focus on the exact memorization behaviors [13, 14, 36, 50, 60]. For example, it has been observed that GPT-2 memorizes long repeated strings such as famous speeches (e.g., Gettysburg Address) [50]. However, once the model drifts from the repeated strings (typically within 100-200 tokens), it displays widening diversity [50]. Tirumala et al. [60] show that language models memorize the training data before over-fitting and nouns and numbers are memorized first. McCoy et al. [46] shows language models can memorize very long sequences with over 1000 words from the training data. Carlini et al. [15] shows it is possible to extract training data by querying language models and demonstrate the training data extraction attack [15] and the membership inference attack [13] on GPT-2 [50].

[7] https://github.com/stanford-crfm/mistral

[8] https://huggingface.co/docs/transformers/model_doc/gpt_neo

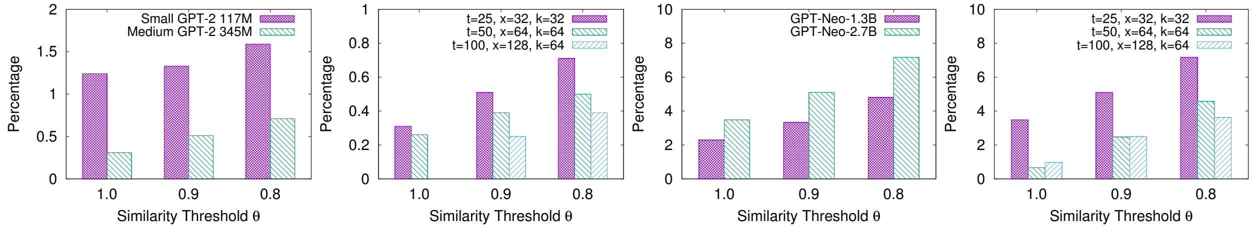
[9] https://c4-search.apps.allenai.org/



(a) OpenWebText (8M texts, $t = 50$) (b) OpenWebText (8M texts, $t = 50$) (c) OpenWebText ($k = 32$, $\theta = 0.9$) (d) OpenWebText ($k = 32$, $\theta = 0.9$)

(e) Pile (825 GB, $t = 50$) (f) Pile (825 GB, $t = 50$) (g) Pile ($k = 32$, $\theta = 0.9$) (h) Pile ($k = 32$, $t = 50$)

Figure 3: Evaluating Query Processing.



(a) $t = 25$, $x = 32$, $k = 32$ (b) Medium GPT-2 345M Model (c) $t = 25$, $x = 32$, $k = 32$ (d) GPT-Neo-2.7B Model

Figure 4: Evaluating Language Model Memorization.

Table 1: Examples of generated texts (query sequences) and their near-duplicate sequences in the training corpus Pile.

Generated Text	Training Text
Copyright (C) 2016 Turi\n *\n * This program is free software: you can redistribute it and/or modify\n * it under the terms of the GNU General Public License as published by\n * the Free Software Foundation, either version 3 of the License, or\n * (at your more details.\n *\n * You should have received a copy of the GNU General Public License\n * along with this program. If not, see <http://www.gnu.org/licenses/>.\n */\n#ifndef GLSUB_BINARY_H\n#define GLS	Copyright 2016 by Sehraf*\n *\n * This program is free software: you can redistribute it and/or modify*\n * it under the terms of the GNU Lesser General Public License as *\n * published by the Free Software Foundation, either version 3 of the *\n * License, or (at your option) any later version. *\n * See the GNU General Public License for\n * more details.\n *\n * You should have received a copy of the GNU General Public License along\n * with this program. If not, see <http://www.gnu.org/licenses/>.\n */\n\n#ifndef TRINITY_AREA_BOUNDARY_H\n#define TRINITY_AREA_BOUNDARY_H\n
UNPUBLISHED\n\n UNITED STATES COURT OF APPEALS\n FOR THE FOURTH CIRCUIT\n\n\n No. 09-4269\n\n\nUNITED STATES OF AMERICA,\n\n Plaintiff - Appellee,\n\n v.\n\nTHOMAS JOHNSON,\n\n Defendant - Appell	UNPUBLISHED\n\nUNITED STATES COURT OF APPEALS\nFOR THE FOURTH CIRCUIT\n\n\nNo. 11-4269\n\n\nUNITED STATES OF AMERICA,\n\nPlaintiff - Appellee,\n\nv.\n\nJOHN MOWAD JOHNSON,\n\Defendant -

Lee et al. shows that over 1% of tokens generated unprompted by a language model are part of a memorized sequence and deduplicating training data offers significant advantages (including reducing memorization) and no observed disadvantages to language modeling [39]. Kandpal et al. [36] shows that empirically the rate a training sequence is emitted by a language model is superlinear to the sequence's frequency in the training corpus. For example, on average, a sequence that appears 10 times in the training corpus is generated 1000× more often than a unique sequence in the training corpus. At the same time, Carlini et al. [14] found that the chance language models emit memorized training data significantly (superlinearly) grows when the model size, the sequence's frequency in the training corpus, or the context length increases.

7 Conclusion

In this paper, we study how many texts generated by large neural language models have near-duplicates in the training corpus. However, as modern language models are trained on larger and larger corpora (up to 1 terabytes) and the number of sequences in a text is quadratic to the text length, it is computational challenge to search near-duplicates in the large-scale text corpus. To address this issue, we develop an efficient and scalable near-duplicate sequence search algorithm based on the min-hash techniques. Experimental results show that our algorithm achieved high performance and good scalability.

References

[1] E. Agirre, C. Banea, D. M. Cer, M. T. Diab, A. Gonzalez-Agirre, R. Mihalcea, G. Rigau, and J. Wiebe. Semeval-2016 task 1: Semantic textual similarity, monolingual and cross-lingual evaluation. In *Proceedings of the 10th International Workshop on Semantic Evaluation*, pages 497–511. The Association for Computer Linguistics, 2016.

[2] S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe. Nearest common ancestors: A survey and a new algorithm for a distributed environment. *Theory Comput. Syst.*, 37(3):441–456, 2004.

[3] M. Artetxe, G. Labaka, E. Agirre, and K. Cho. Unsupervised neural machine translation. *arXiv preprint arXiv:1710.11041*, 2017.

[4] R. Baeza-Yates, B. Ribeiro-Neto, et al. *Modern information retrieval*, volume 463. ACM press New York, 1999.

[5] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval.* ACM Press / Addison-Wesley, 1999.

[6] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, pages 131–140, 2007.

[7] A. Behm, C. Li, and M. J. Carey. Answering approximate string queries on large data sets using external memory. In *ICDE*, pages 888–899, 2011.

[8] M. A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *J. Algorithms*, 57(2):75–94, 2005.

[9] S. Brin, J. Davis, and H. Garcia-Molina. Copy detection mechanisms for digital documents. In *SIGMOD*, pages 398–409. ACM Press, 1995.

[10] A. Z. Broder. On the resemblance and containment of documents. In *SEQUENCES*, pages 21–29. IEEE, 1997.

[11] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. *Comput. Networks*, 29(8-13):1157–1166, 1997.

[12] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[13] N. Carlini, S. Chien, M. Nasr, S. Song, A. Terzis, and F. Tramèr. Membership inference attacks from first principles. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, pages 1897–1914. IEEE, 2022.

[14] N. Carlini, D. Ippolito, M. Jagielski, K. Lee, F. Tramèr, and C. Zhang. Quantifying memorization across neural language models. *CoRR*, abs/2202.07646, 2022.

[15] N. Carlini, F. Tramèr, E. Wallace, M. Jagielski, A. Herbert-Voss, K. Lee, A. Roberts, T. B. Brown, D. Song, Ú. Erlingsson, A. Oprea, and C. Raffel. Extracting training data from large language models. In M. Bailey and R. Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 2633–2650. USENIX Association, 2021.

[16] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

[17] A. Chowdhury, O. Frieder, D. Grossman, and M. C. McCabe. Collection statistics for fast duplicate document detection. *ACM Transactions on Information Systems (TOIS)*, 20(2):171–191, 2002.

[18] J. G. Conrad, X. S. Guo, and C. P. Schriber. Online duplicate document detection: signature reliability in a dynamic retrieval environment. In *Proceedings of the twelfth international conference on Information and knowledge management*, pages 443–452, 2003.

[19] D. Deng, G. Li, and J. Feng. A pivotal prefix based filtering algorithm for string similarity search. In *SIGMOD Conference*, pages 673–684, 2014.

[20] D. Deng, G. Li, H. Wen, and J. Feng. An efficient partition based method for exact set similarity joins. *Proc. VLDB Endow.*, 9(4):360–371, 2015.

[21] D. Deng, G. Li, H. Wen, H. V. Jagadish, and J. Feng. META: an efficient matching-based method for error-tolerant autocompletion. *PVLDB*, 9(10):828–839, 2016.

[22] D. Deng, Y. Tao, and G. Li. Overlap set similarity joins with theoretical guarantees. In *SIGMOD*, pages 905–920. ACM, 2018.

[23] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[24] A. Fan, M. Lewis, and Y. N. Dauphin. Hierarchical neural story generation. In I. Gurevych and Y. Miyao, editors, *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*, pages 889–898. Association for Computational Linguistics, 2018.

[25] W. Feng and D. Deng. Allign: Aligning all-pair near-duplicate passages in long texts. In G. Li, Z. Li, S. Idreos, and D. Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 541–553. ACM, 2021.

[26] J. Fischer. Optimal succinctness for range minimum queries. In A. López-Ortiz, editor, *LATIN 2010: Theoretical Informatics, 9th Latin American Symposium, Oaxaca, Mexico, April 19-23, 2010. Proceedings*, volume 6034 of *Lecture Notes in Computer Science*, pages 158–169. Springer, 2010.

[27] P. Gage. A new algorithm for data compression. *C Users J.*, 12(2):23âĂŞ38, feb 1994.

[28] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima, S. Presser, and C. Leahy. The pile: An 800gb dataset of diverse text for language modeling. *CoRR*, abs/2101.00027, 2021.

[29] R. Gheorghe, M. L. Hinman, and R. Russo. *Elasticsearch in action.* Manning Shelter Island, NY, 2015.

[30] A. Gokaslan and V. Cohen. Openwebtext corpus.

[31] A. Halavais. *Search engine society.* John Wiley & Sons, 2017.

[32] O. A. Hamid, B. Behzadi, S. Christoph, and M. R. Henzinger. Detecting the origin of text segments efficiently. In *WWW*, pages 61–70. ACM, 2009.

[33] J. R. Hamilton and T. K. Nayak. Microsoft sql server full-text search. *IEEE Data Eng. Bull.*, 24(4):7–10, 2001.

[34] T. C. Hoad and J. Zobel. Methods for identifying versioned and plagiarized documents. *J. Assoc. Inf. Sci. Technol.*, 54(3):203–215, 2003.

[35] S. Ji, G. Li, C. Li, and J. Feng. Efficient interactive fuzzy keyword search. In *WWW*, pages 433–439, 2009.

[36] N. Kandpal, E. Wallace, and C. Raffel. Deduplicating training data mitigates privacy risks in language models. In K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvári, G. Niu, and S. Sabato, editors, *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162 of *Proceedings of Machine Learning Research*, pages 10697–10707. PMLR, 2022.

[37] J. W. Kim, K. S. Candan, and J. Tatemura. Efficient overlap and content reuse detection in blogs and online news articles. In *WWW*, pages 81–90. ACM, 2009.

[38] T. Kwiatkowski, J. Palomaki, O. Redfield, M. Collins, A. Parikh, C. Alberti, D. Epstein, I. Polosukhin, J. Devlin, K. Lee, et al. Natural questions: a benchmark for question answering research. *Transactions of the Association for Computational Linguistics*, 7:453–466, 2019.

[39] K. Lee, D. Ippolito, A. Nystrom, C. Zhang, D. Eck, C. Callison-Burch, and N. Carlini. Deduplicating training data makes language models better. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8424–8445, 2022.

[40] J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of Massive Datasets, 2nd Ed.* Cambridge University Press, 2014.

[41] G. Li, D. Deng, and J. Feng. Faerie: efficient filtering algorithms for approximate dictionary-based entity extraction. In *SIGMOD Conference*, pages 529–540, 2011.

[42] G. Li, D. Deng, J. Wang, and J. Feng. PASS-JOIN: A partition-based method for similarity joins. *PVLDB*, 5(3):253–264, 2011.

[43] G. Li, S. Ji, C. Li, and J. Feng. Efficient fuzzy full-text type-ahead search. *VLDB J.*, 20(4):617–640, 2011.

[44] P. Li, A. B. Owen, and C. Zhang. One permutation hashing. In *NIPS*, pages 3122–3130, 2012.

[45] U. Manber. Finding similar files in a large file system. In *USENIX Winter 1994 Technical Conference*, pages 1–10. USENIX Association, 1994.

[46] R. T. McCoy, P. Smolensky, T. Linzen, J. Gao, and A. Celikyilmaz. How much do language models copy from their training data? evaluating linguistic novelty in text generation using RAVEN. *CoRR*, abs/2111.09509, 2021.

[47] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur. Recurrent neural network based language model. In *Interspeech*, volume 2, pages 1045–1048. Makuhari, 2010.

[48] M. Potthast, A. Barrón-Cedeño, A. Eiselt, B. Stein, and P. Rosso. Overview of the 2nd international competition on plagiarism detection. In *CLEF 2010 LABs and Workshops, Notebook Papers*, volume 1176 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2010.

[49] D. M. Powers. Applications and explanations of zipfâĂŹs law. In *New methods in language processing and computational natural language learning*, 1998.

[50] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[51] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv e-prints*, 2019.

[52] R. Ramakrishnan, J. Gehrke, and J. Gehrke. *Database management systems*, volume 3. McGraw-Hill New York, 2003.

[53] J. Sarzynska-Wawer, A. Wawer, A. Pawlak, J. Szymanowska, I. Stefaniak, M. Jarkiewicz, and L. Okruszek. Detecting formal thought disorder by deep contextualized word representations. *Psychiatry Research*, 304:114135, 2021.

[54] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *SIGMOD*, pages 76–85. ACM, 2003.

[55] A. See, P. J. Liu, and C. D. Manning. Get to the point: Summarization with pointer-generator networks. *arXiv preprint arXiv:1704.04368*, 2017.

[56] J. Seo and W. B. Croft. Local text reuse detection. In *SIGIR*, pages 571–578. ACM, 2008.

[57] A. Shatdal and J. F. Naughton. Adaptive parallel aggregation algorithms. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, page 104âĂŞ114, New York, NY, USA, 1995. Association for Computing Machinery.

[58] N. Shivakumar and H. Garcia-Molina. Finding near-replicas of documents and servers on the web. In *The World Wide Web and Databases, International Workshop WebDB'98*, volume 1590 of *Lecture Notes in Computer Science*, pages 204–212. Springer, 1998.

[59] M. Thorup. Bottom-k and priority sampling, set similarity and subset sums with minimal independence. In D. Boneh, T. Roughgarden, and J. Feigenbaum, editors, *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 371–380. ACM, 2013.

[60] K. Tirumala, A. H. Markosyan, L. Zettlemoyer, and A. Aghajanyan. Memorization without overfitting: Analyzing the training dynamics of large language models. *CoRR*, abs/2205.10770, 2022.

[61] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[62] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *SIGMOD*, pages 85–96, 2012.

[63] P. Wang, C. Xiao, J. Qin, W. Wang, X. Zhang, and Y. Ishikawa. Local similarity search for unstructured text. In *SIGMOD*, pages 1991–2005. ACM, 2016.

[64] Z. Wang, C. Zuo, and D. Deng. Txtalign: Efficient near-duplicate text alignment search via bottom-k sketches for plagiarism detection. In Z. Ives, A. Bonifati, and A. E. Abbadi, editors, *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 1146–1159. ACM, 2022.

[65] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang. Efficient similarity joins for near-duplicate detection. *ACM Trans. Database Syst.*, 36(3):15, 2011.

[66] H. Yang and J. Callan. Near-duplicate detection for erulemaking. In *Proceedings of the 2005 national conference on Digital government research*, pages 78–86, 2005.

[67] H. Yang and J. Callan. Near-duplicate detection by instance-level constrained clustering. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 421–428, 2006.

[68] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le. Xlnet: Generalized autoregressive pretraining for language understanding. *Advances in neural information processing systems*, 32, 2019.