

Lecture 3: Pre-training Data Processing

Instructors: *Alex Andoni, Edo Liberty*Scribes: *Varun Mangla, Xinyu Lu*

Part I

Covered in Class

1 Introduction

In this lecture, the data used for training Large Language Models is discussed, with emphasis on how data can be selected, models can be encouraged to focus on "good" sources, and de-duplication of resources can be done.

Additional sections on data selection and MinHash deduplication will be provided to further explore concepts related to the lecture.

2 Data Selection

In the previous lecture, we discussed how tokenization can be used to transform data into a form that models can interpret and use through methods such as Byte-Pair encoding. We now shift our focus to a related problem, that being how we select data to tokenize.

2.1 Crawling the Internet

One naive approach may be to simply crawl the entire internet, and train a model on the resulting corpus of documents produced. We can refer to models such as Llama 3, an open-source model trained on a corpus of $1.5 * 10^{13}$ tokens sourced from an input roughly 10 times larger, as evidence that feeding large amounts of data into a model is a worthwhile investment. However, while it is ideal to train models with as much data as possible, this can lead to poor data quality, which has adverse effects on models¹.

Therefore, we must refine the internet data received after crawling, which can amount to approximately 10^{14} - 10^{15} bytes in 10^9 - 10^{10} documents, through a multi-step process.

2.1.1 Throwing out Data

One step we can take is to simply throw out all data from "bad" websites, such as those that we know to contain false information or ideas that we don't want our model to learn. This can extend to spam data or any other sort of data that one wishes to filter out prior to further processing.

¹See [7], which details this for tabular data.

2.1.2 Partitioning by Language

Then, we can partition the data by the language used in it, which can be done using a smaller model specifically trained for the language identification task. This model, which is a classifier, will group documents of primarily a certain language together. A smaller model is used due to the large scale of the data; given that there can be approximately 10^{10} documents, our main goal is to process all of them within a reasonable amount of time, hence the need for a relatively small model or algorithm.

2.1.3 Quality Filtering

If a document is too long, it may not be a standard piece of text, such as a machine-produced log. Furthermore, code and other non-natural forms of writing can be separated from traditional texts in this step.

2.1.4 Deduplication

Then, we wish to ensure that duplicate documents are eliminated from the data. This has two main benefits:

- **Reducing the size of the data:** This makes training more efficient, as there are fewer documents in the data, meaning that the model will have to iterate over less in its training steps.
- **Improving the quality of training:** When a model is being trained, it iterates over each document in the data only some fixed number of times, which is to ensure that it does not overfit to any specific document². However, if a document repeats multiple times, it may be iterated on a disproportionate amount of times compared to other, non-duplicated documents. One goal of de-duplication is to ensure that this case doesn't occur such that the model doesn't overfit to a subset of documents that repeat a large amount of times.

2.1.5 Selection

Of the documents remaining, they have a varying level of "quality"; documents sourced from Wikipedia and Arxiv may have a relatively high amount of quality, while some documents from less-reputable websites may be of poor quality. In this step, we wish to ensure that the data mainly comes from the distribution of "good" sources, which is done through resampling the internet data around it.

3 Deduplication

This is a classic problem that has existed prior to LLMs, and many algorithms exist to tackle it. There are two main types of deduplication: exact and near-match.

²This would result in the model not learning the underlying structure of the document, but just simply memorizing the text inside of it.

3.1 Exact Deduplication

This step removes long verbatim repeats. On the web, whole pages are rarely identical (site names, copyright footers, etc. differ), so we deduplicate repeated *substrings* rather than whole documents. We first fix the problem precisely.

Definition 1 (Exact deduplication). *Given $d \in \Sigma^n$ (all documents concatenated) and a window length k (e.g. $k = 50$), let $w_j = d[j..j+k-1]$ for $j = 1, \dots, n-k+1$. Window j is a repeat if $w_j = w_{j'}$ for some $j' < j$. Output $P = \bigcup_{j \text{ repeat}} \{j, \dots, j+k-1\}$, the positions covered by repeat windows; deleting P keeps the first copy of every repeated substring.*

3.1.1 Solution 1

Store all length k substrings in a hash table. If a string is found to collide with an existing string in the hash table, throw it out.

The most direct approach is to hash every length- k window and look for matches in a hash table. Fix a hash function $h : \Sigma^k \rightarrow [M]$ and scan $j = 1, 2, \dots$: compute $h(w_j)$ and look at that bucket, which stores positions of earlier distinct windows with the same hash value. Compare w_j character by character with each stored window. If one matches, j is a repeat; otherwise insert j into the bucket. The comparison is necessary: equal hash values do not imply equal strings, so treating a collision as a repeat would delete distinct text.

The next claim records what this costs. It separates two issues: *correctness*, which is guaranteed by the exact comparison, and *running time*, which depends on how many wasted comparisons are caused by collisions between distinct windows.

Claim 2. *If h is drawn from a universal family ($\Pr[h(u) = h(v)] \leq 1/M$ for $u \neq v$) with $M \geq n^2$, Solution 1 is always correct and runs in expected time $O(nk)$, storing one entry per distinct window.*

Proof. j is declared a repeat only after an exact match with an earlier window; conversely if $w_j = w_{j'}$, $j' < j$, the first occurrence of that string is in the table, in the same bucket, so it is found. Time: hashing w_j costs $O(k)$, and each comparison costs $O(k)$. Each lookup makes at most one successful comparison, plus one failed comparison per distinct stored window colliding with w_j . The expected number of colliding pairs of distinct windows is at most $\binom{n}{2}/M \leq 1/2$, so the expected number of failed comparisons is $O(1)$ in total. Hence the expected time is $O(nk)$. $\square$

Here k is the window length, and computing one hash from scratch costs $O(k)$.

Time: $O(nk)$.

Space: about 8–16 bytes per token. With $k = 50$, it is akin to 50 passes over the corpus, so it is not ideal for large n . The wasted work is in recomputing each hash from scratch, which the next solution avoids.

3.1.2 Solution 1.5

Given the large runtime of Solution 1, we hope to make improvements where we can; this can be done through more efficiently computing hashes through streaming. In this approach, a sliding window can be constructed of length- k substrings across n , where computing the next hash from the previous one takes $O(1)$ time. This is expressed graphically below:

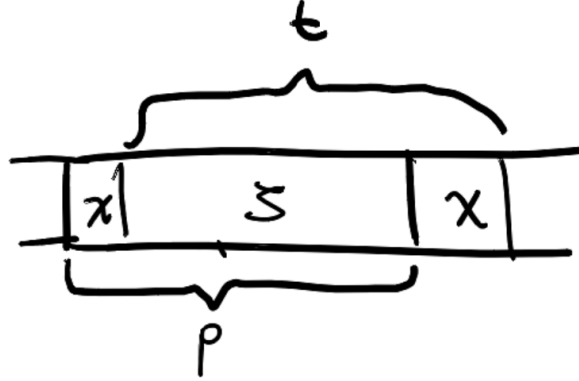


Figure 1: For length- k substrings p and t , we can do $h(p) \rightarrow h(t)$ in $O(1)$ time.

Consecutive windows overlap in $k - 1$ symbols, so a hash with the right algebraic structure can be updated in $O(1)$ time instead of recomputed. We use a polynomial hash. Identify Σ with $\{0, \dots, |\Sigma| - 1\}$, fix a prime $P > |\Sigma|$ and a uniformly random $B \in \{0, \dots, P - 1\}$, and set $H(w) = \sum_{i=1}^k w[i]B^{k-i} \bmod P$. We need two properties: the hash must be updatable in $O(1)$ (this gives speed), and distinct windows must rarely collide (this gives reliability, since we will now rely on hash equality).

Lemma 3. *This lemma consists of two parts.*

- (a) *If w_{j+1} is obtained from w_j by removing its first character x and appending x' , then $H(w_{j+1}) = (H(w_j) - xB^{k-1})B + x' \bmod P$, which takes $O(1)$ operations given B^{k-1} .*
- (b) *For distinct $u, v \in \Sigma^k$, $\Pr_B[H(u) = H(v)] \leq (k - 1)/P$.*

Proof. We individually prove parts (a) and (b) of the above lemma.

(a) Consider $w_j = xs$. Then $H(w_j) - xB^{k-1} = \sum_{i=1}^{k-1} s[i]B^{k-1-i}$; multiplying by B and adding x' gives $H(sx')$.

(b) $Q(z) = \sum_i (u[i] - v[i])z^{k-i}$ is a nonzero polynomial over $\mathbb{F}_P$ of degree $\leq k - 1$ (symbols are distinct residues since $P > |\Sigma|$), so it has at most $k - 1$ roots; $H(u) = H(v)$ iff $Q(B) = 0$. $\square$

Part (a) of the lemma provides the speed and part (b) the reliability. The next claim combines them: we choose P large enough that a union bound over *all* pairs of windows still leaves a small total failure probability.

Claim 4. *Take a prime $P \geq n^3k$. With probability $\geq 1 - 1/n$, distinct windows have distinct H -values, so declaring j a repeat iff $H(w_j)$ was already seen finds exactly the repeat windows, in $O(n)$ time.*

Proof. By Lemma 3(b) and a union bound over $\leq n^2/2$ pairs, some pair collides with probability $\leq \frac{n^2}{2} \cdot \frac{k-1}{P} \leq \frac{1}{n}$. Otherwise H -equality is string equality. Computing all $H(w_j)$ takes $O(k)$ for w_1 and $O(1)$ each afterwards (Lemma 3(a)); residues have $O(\log n)$ bits, so computation is $O(1)$. $\square$

This is a Monte Carlo algorithm. A zero-error version verifies each detected match as in Solution 1, at extra cost $O(k)$ per match.

Time: $O(n)$.

Space: same as Solution 1, which is too large for a big corpus (8–16 bytes per token).

3.1.3 Solution 2

Therefore, to alleviate the storage problem while also maintaining $O(n)$ runtime, we wish to do exact deduplication approximately, such that substrings are mistakenly labeled as repeats with some probability, using a bloom filter.

We only need a data structure that supports “insert” and “have we (probably) seen this before?”. A Bloom filter answers this with far fewer bits than storing the items, at the price of occasional false “Yes” answers. For deduplication, a false positive means we delete a window that was in fact unique.

Definition 5 (Bloom filter). *A Bloom filter is a bit array $T \in \{0, 1\}^m$ (initially all 0, $m = \Theta(n)$) with l fully random hash functions $g_1, \dots, g_l : U \rightarrow [m]$, where U is the universe of fingerprints (e.g. $x = H(w)$). $\text{INSERT}(x)$ sets $T[g_i(x)] \leftarrow 1$ for all $i \in [l]$. $\text{QUERY}(x)$ outputs “Yes” iff $T[g_i(x)] = 1$ for all $i \in [l]$.*

For deduplication, scan the windows in order; for each window compute $x = H(w_j)$. If $\text{QUERY}(x)$ says “Yes”, declare j a repeat; otherwise $\text{INSERT}(x)$.

Algorithm 1 Solution 2: Bloom filter

```

 $T \leftarrow 0^m$ 
procedure  $\text{INSERT}(x)$ 
  for  $i \leftarrow 1$  to  $l$  do
     $T[g_i(x)] \leftarrow 1$ 
  end for
end procedure
function  $\text{QUERY}(x)$ 
  if  $T[g_i(x)] = 1$  for all  $i \in [l]$  then
    return “Yes”
  else
    return “No”
  end if
end function

```

Before relying on the Bloom filter, we must first understand how often it fails. The following claim states that the bloom filter algorithm never misses a genuine repeat (no false negatives), and gives the false positive probability in a form we can then optimize over l and m .

Claim 6. *Suppose $N \leq n$ distinct elements were inserted.*

- (a) *If x was inserted, $\text{QUERY}(x)$ says “Yes”.*
- (b) *If x was not inserted, let Y be the fraction of ones in T ; then $\Pr[\text{“Yes”}] = \mathbb{E}[Y^l]$ and $\mathbb{E}[Y] = 1 - (1 - \frac{1}{m})^{lN}$.*
- (c) *For $m = \Theta(n)$, $\Pr[\text{“Yes”}] \approx (1 - e^{-lN/m})^l$.*

Proof. We wish to first condition on the table, so that the l probes are independent coin flips with bias Y , then compute $\mathbb{E}[Y]$ and argue that Y is concentrated.

(a) INSERT set all probed bits to 1, and bits are never reset.

(b) Fix T , which depends only on the g_i at inserted elements. Since x is not inserted, $g_1(x), \dots, g_l(x)$ are independent and uniform on $[m]$, independent of T , so each probe hits a 1 with probability Y and $\Pr[\text{Yes} \mid T] = Y^l$. A fixed bit is 0 iff none of the lN independent uniform values $g_i(y)$ (y inserted) lands on it, which has probability $(1 - \frac{1}{m})^{lN}$; so $\mathbb{E}[Y] = 1 - (1 - \frac{1}{m})^{lN}$.

(c) Since $1 - z \approx e^{-z}$ for small z , $\mathbb{E}[Y] \approx 1 - e^{-lN/m}$. (This step is an approximation, not an upper bound, since $(1 - \frac{1}{m})^{lN} \leq e^{-lN/m}$.) Also Y is concentrated: the number of ones is a function of lN independent values, each changing it by at most 1, so by McDiarmid's inequality $\Pr[|Y - \mathbb{E}Y| \geq \varepsilon] \leq 2\exp(-2\varepsilon^2 m^2 / (lN))$, which is exponentially small when $lN = O(m)$. Hence $\mathbb{E}[Y^l] \approx (\mathbb{E}Y)^l$. $\square$

Claim 6 leaves l and m free. The next claim fixes $m = 8n$ (one byte per item, independent of the corpus) and finds the best l , which gives the concrete error rate.

Claim 7. *Let $m = 8n$ and $N \leq n$. (a) $F(l) = (1 - e^{-l/8})^l$ is minimized over real l at $8\ln 2 \approx 5.55$, and over integers at $l = 6$ ($F(6) \approx 0.02158 < F(5) \approx 0.02168$). (b) For $l = 6$ and $n \geq 10^9$, $\Pr[\text{false positive}] \leq 2.2\%$, where a false positive means $\text{QUERY}(x)$ says “Yes” for an x that was not inserted.*

Proof. In this proof, part (a) is a calculus argument on the approximate error rate; part (b) turns the approximation of Claim 6(c) into a rigorous bound, using the two ingredients from its proof (a two-sided estimate of $\mathbb{E}[Y]$ and concentration of Y).

(a) Let $c = 8$, $u = e^{-l/c}$, $g(l) = \ln F = l \ln(1 - u)$. Then $g'(l) = \ln(1 - u) - \frac{u \ln u}{1 - u} = \frac{h(u)}{1 - u}$ with $h(u) = (1 - u) \ln(1 - u) - u \ln u$. We have $h(0^+) = h(\frac{1}{2}) = h(1^-) = 0$ and $h''(u) = \frac{1}{1 - u} - \frac{1}{u}$, negative on $(0, \frac{1}{2})$ and positive on $(\frac{1}{2}, 1)$. So $h > 0$ on $(0, \frac{1}{2})$ (strictly concave, zero endpoints) and $h < 0$ on $(\frac{1}{2}, 1)$ (strictly convex, zero endpoints). As $u < \frac{1}{2}$ iff $l > c \ln 2$, g decreases for $l < c \ln 2$ and increases after. So the integer optimum is 5 or 6; direct evaluation gives $F(5) \approx 0.02168$, $F(6) \approx 0.02158$.

(b) The false positive probability is non-decreasing in N , so take $N = n$, where $lN/m = 0.75$. Using $\ln(1 - z) \geq -z - z^2$ for $z \leq \frac{1}{2}$: $\mathbb{E}[Y] \leq 1 - e^{-0.75(1+1/m)} \leq 0.5277$ for $n \geq 10^3$. McDiarmid with $\varepsilon = 10^{-3}$ gives $\Pr[Y > 0.5287] \leq 2e^{-21000}$ for $n \geq 10^9$. Hence $\mathbb{E}[Y^6] \leq 0.5287^6 + 2e^{-21000} < 0.0219 < 2.2\%$. $\square$

Time: $O(n)$.

Space: $8n$ bits, since $m = 8n$.

This serves as a substantial improvement over both Solutions 1 and 1.5, maintaining the $O(n)$ time of the latter solution while significantly improving on the space bound for a large corpus.

3.1.4 Solution 3

Another approach is deterministic: sort all suffixes of the corpus, so that repeated substrings end up next to each other. Append a symbol $\$$ smaller than every symbol of Σ to d , and let $s_i = d[i..n + 1]$ be the suffix starting at i ; this makes all suffixes distinct and no suffix a prefix of another.

Definition 8 (Suffix array, LCP array). *The suffix array $SA[1..n + 1]$ lists the starting positions of the suffixes in lexicographic order: $s_{SA[1]} < s_{SA[2]} < \dots$. Let $\text{rank}(i)$ be the r with $SA[r] = i$. Let $\text{lcp}(x, y)$ be the length of the longest common prefix of strings x, y . The LCP array is $LCP[r] = \text{lcp}(s_{SA[r-1]}, s_{SA[r]})$ for $r \geq 2$, and $LCP[1] = 0$.*

The LCP array only compares *neighbors* in sorted order, but we want to know about repeats between arbitrary pairs of suffixes. The lemma below bridges the two: if two suffixes share a long prefix, then so does everything sorted between them, so the lcp of any pair is the minimum of the neighbor values in between.

Lemma 9. (a) *For strings $x < y < z$: $\text{lcp}(x, z) = \min(\text{lcp}(x, y), \text{lcp}(y, z))$.* (b) *For ranks $a < b$: $\text{lcp}(s_{SA[a]}, s_{SA[b]}) = \min_{a < r \leq b} LCP[r]$.*

Proof. (a) is the local statement that a string sorted between two strings with a common prefix must have that prefix; (b) follows by induction.

(a) Let $\ell = \text{lcp}(x, z)$ and q their common prefix of length ℓ . If y did not start with q , let $t \leq \ell$ be the first position where y differs from q (or ends). Then y agrees with x, z before t , and $y[t] < q[t]$ (or y ends) gives $y < x$, while $y[t] > q[t]$ gives $y > z$; both are contradictions. So y starts with q , and the minimum is $\geq \ell$. Conversely, if x and z both share $\lambda = \min(\cdot, \cdot)$ characters with y , they share them with each other, so $\ell \geq \lambda$.

(b) By induction on $b - a$ using (a) with $y = s_{SA[b-1]}$; the case $b = a + 1$ is the definition of LCP . $\square$

With the lemma we can characterize repeated windows using only sorted neighbors, which is why a single linear scan will suffice.

Claim 10. *Let $r = \text{rank}(p)$ and $p \leq n - k + 1$. The window w_p occurs at another position iff $LCP[r] \geq k$ or $LCP[r + 1] \geq k$ (with $LCP[n + 2] := 0$).*

Proof. Recall “if” is immediate from the definition of LCP ; “only if” uses Lemma 9(b) to push the information from an arbitrary occurrence to an adjacent rank. ($\Leftarrow$) If $LCP[r] \geq k$, the predecessor suffix s_q agrees with s_p on k characters, so $w_q = w_p$ with $q \neq p$ (the common prefix cannot contain \$, which occurs once); symmetric for $LCP[r + 1]$. ($\Rightarrow$) If $w_q = w_p$ with $q \neq p$, let $r' = \text{rank}(q)$; then $\text{lcp}(s_p, s_q) \geq k$. If $r' > r$, Lemma 9 gives $\min_{r < t \leq r'} LCP[t] \geq k$, so $LCP[r + 1] \geq k$; if $r' < r$, then $LCP[r] \geq k$ likewise. $\square$

To keep the first copy (as in Definition 1), in each maximal run of consecutive ranks linked by $LCP \geq k$ keep only the smallest position and mark the others as repeats.

The claim turns deduplication into a scan over the LCP array. What remains is to build SA and LCP fast: naively sorting n suffixes with comparisons that may each take $\Theta(n)$ time would be far too slow.

Theorem 11. *SA and LCP can be computed in $O(n)$ time, hence the exact deduplication problem (Definition 1) is solved exactly in $O(n)$ time.*

Proof. We cite linear-time suffix array construction, and prove the LCP construction by an amortized argument. Linear-time suffix array construction is a known result (DC3 of Kärkkäinen–Sanders [6]; SA-IS of Nong–Zhang–Chan [8]), which we cite without proof. For LCP we prove Kasai et al.’s algorithm. Let $\pi(i)$ be the suffix just before s_i in sorted order and $\lambda_i = \text{lcp}(s_i, s_{\pi(i)})$. Recall $\lambda_{i+1} \geq \lambda_i - 1$. If $\lambda_i \leq 1$ this is trivial. Otherwise, with $j = \pi(i)$, both s_j, s_i start with the same character and $s_j < s_i$, so $s_{j+1} < s_{i+1}$ and $\text{lcp}(s_{j+1}, s_{i+1}) = \lambda_i - 1$. By Lemma 9(b), every LCP entry with rank in $(\text{rank}(j + 1), \text{rank}(i + 1)]$ is at least $\lambda_i - 1$; in particular $LCP[\text{rank}(i + 1)] = \lambda_{i+1} \geq \lambda_i - 1$. So computing λ_{i+1} can start comparing at offset $\lambda_i - 1$. Processing $i = 1, \dots, n$ in text order, the running value decreases by at most 1 per step and never exceeds n , so the total number of character comparisons is at most $2n = O(n)$. The scan of Claim 10 and the difference array are $O(n)$. $\square$

Time: $O(n)$.

Space: $O(n)$ bytes, i.e. $O(n \log n)$ bits (the suffix array and LCP array).

3.2 Near-Match Deduplication

In Near-Match deduplication, we want to remove documents that have very similar text with small modifications.

3.2.1 Problem

Exact methods only catch verbatim repeats. To catch documents that are equal up to small edits, we first need a precise notion of similarity. We represent a document by its set of length- k substrings (shingles): a small edit only changes at most k shingles per changed symbol, so heavily overlapping documents have heavily overlapping shingle sets.

Definition 12 (Shingle set). *For a fixed k , the shingle set of a document x is the set $A(x) \subseteq \Sigma^k$ of all length- k substrings of x (multiplicities and order are discarded).*

Example 13. *For $k = 5$, “banana” has $A = \{\text{banan}, \text{anana}\}$.*

We compare two shingle sets by the fraction of shingles they share.

Definition 14 (Jaccard similarity). *For nonempty A, B , $J(A, B) = \frac{|A \cap B|}{|A \cup B|} \in [0, 1]$; it is 1 iff $A = B$ and 0 iff they are disjoint.*

Documents are near-duplicates if $J(A, B) \geq \tau$ for a chosen threshold τ (e.g. $\tau = 0.75$); we then remove one of the two.

3.2.2 Naive Approach

One approach is, for each document, to go through each other document and compare the Jaccard similarity between the two to see if they are near-matches. If this is true, remove the second document from the data.

Time: $O(n^2 * m)$, where n is the number of documents, and m is the time to compute one Jaccard similarity (proportional to the shingle-set sizes), which is far too slow for $n \approx 10^{10}$ documents. We therefore first compress each document into a short sketch.

3.2.3 Better Approach

In this approach, we aim to speed up the computation of the Jaccard for each pair of documents through sketching.

Definition 15 (Sketching). *A sketching scheme with parameters ε, δ is a randomized map f from sets to b -bit strings and an estimator $R : \{0, 1\}^b \times \{0, 1\}^b \rightarrow [0, 1]$ such that for all A, B : $\Pr[|R(f(A), f(B)) - J(A, B)| \leq \varepsilon] \geq 1 - \delta$. The randomness of f is shared by all documents.*

The classical sketch for Jaccard similarity is MinHash. Its building block is a single hash value whose collision probability is exactly the Jaccard similarity.

Definition 16 (MinHash). *Let $U = \Sigma^k$ and $\pi : U \rightarrow \{1, \dots, |U|\}$ a uniformly random permutation. For nonempty $A \subseteq U$, $h_\pi(A) = \min_{u \in A} \pi(u)$.*

The following claim is the reason MinHash works: it says one MinHash comparison is an unbiased 0/1 estimate of $J(A, B)$.

Claim 17. *For nonempty $A, B \subseteq U$: $\Pr_\pi[h_\pi(A) = h_\pi(B)] = J(A, B)$.*

Proof. Recall that the smallest element of $A \cup B$ under π is uniform on $A \cup B$, and the two minima agree exactly when that element lies in $A \cap B$. Let $S = A \cup B$ and x^* the element of S with smallest π -value (unique, as π is injective).

Step 1: x^* is uniform on S . All $|S|!$ relative orders of S under π are equally likely, and each element is first in exactly $(|S| - 1)!$ of them, so $\Pr[x^* = u] = 1/|S|$.

Step 2: $h_\pi(A) = h_\pi(B)$ iff $x^* \in A \cap B$. If $x^* \in A \cap B$, then x^* attains the minimum over $S \supseteq A, B$ and lies in both, so $h_\pi(A) = h_\pi(B) = \pi(x^*)$. Otherwise x^* is missing from one set, say B (the other case is symmetric); then $x^* \in A$, so $h_\pi(A) = \pi(x^*)$, while every element of B differs from x^* and has a larger π -value, so $h_\pi(B) > h_\pi(A)$. Hence $\Pr[h_\pi(A) = h_\pi(B)] = \Pr[x^* \in A \cap B] = |A \cap B|/|A \cup B|$. $\square$

One MinHash comparison is only a 0/1 signal, which is far too noisy on its own. Averaging independent copies gives an accurate estimate, and the next claim says how many copies suffice; this shows that MinHash meets the definition of a sketching scheme.

Claim 18. *Let $\pi_1, \dots, \pi_t$ be independent, $f(A) = (h_{\pi_1}(A), \dots, h_{\pi_t}(A))$ and R the fraction of coordinates where the two sketches agree. If $t \geq \ln(2/\delta)/(2\varepsilon^2)$ then $\Pr[|R - J(A, B)| \leq \varepsilon] \geq 1 - \delta$.*

Proof. We wish to apply a concentration bound to the average of t independent indicators with mean $J(A, B)$. The indicators $X_s = \mathbf{1}[h_{\pi_s}(A) = h_{\pi_s}(B)]$ are independent with mean $J(A, B)$ (Claim 17) and $R = \frac{1}{t} \sum X_s$. Hoeffding's inequality gives $\Pr[|R - J| > \varepsilon] \leq 2e^{-2t\varepsilon^2} \leq \delta$. $\square$

The sketch length t does not depend on the document length, so one comparison costs $O(t)$. Comparing all pairs is still $O(n^2t)$, quadratic in the number of documents; Xinyu's chapter (LSH) removes this quadratic dependence.

Part II

A Further Analysis of Data Selection in Pre-Training (not covered in class; by *Varun Mangla*)

4 Introduction

In the lecture scribe notes, it was mentioned that selection serves as a crucial step in data processing prior to training; it aims to select data points and resample the data such that it follows the same distribution as data from "good" sources³. As described by Albalak et. al. [1], this can be formalized as follows:

Definition 19 (Data Selection). *The above procedure can be defined as a data selection function ϕ acting on a dataset D_{raw} and objective function f_{obj} such that a model M trained on $D = \phi(D_{raw})$ optimizes $f_{obj}(M)$. Often, ϕ is composed of multiple functions $\phi_j, j = 1...n$ such that each performs a specific form of selection; this allows for further granularity in the selection process.*

Therefore, we can formulate the problem as selecting a proper f_{obj} and $\phi_j, j = 1...n$. We can think of f_{obj} as something set by the researchers creating a model prior to training, with one example being to ensure maximal data selection efficiency such that the corpus is significantly reduced in size while maintaining a high level of quality⁴.

5 Defining Data Quality

A common piece of wisdom is that models are only as good as the data they are trained on; therefore, a primary goal of data selection is to select high quality data such that the resulting model is as effective as possible for a specific task. Due to the subjectivity of this process, as well as the wide range of possible tasks a model could be used for, we will examine the task of producing a large language model with high levels of generalization and natural language output⁵ and a specific methodology, namely GneissWeb.

5.1 GneissWeb

GneissWeb [3] is one such procedure that aims to define what a high-quality dataset looks like; it establishes a set of quality heuristics from which it filters data crawled from the internet. In the method, quality is denoted as a recipe using a set of selectors ϕ on the dataset, two of which are as follows.

5.1.1 Document Classification

One technique used by GneissWeb is implementing a set of smaller classifier models trained on some subset of high and low-quality data ($n_{documents} < 500,000$) from a pre-annotated dataset⁶, and then using them to assign quality scores $\in [0, 1]$ to the entire dataset D of interest.

³Examples include Arxiv and Wikipedia, though this can vary due to the subjective nature of this procedure.

⁴This can vary depending on what stage of training the model is in, though we only concern ourself with pre-training.

⁵Think of ChatGPT, Claude, and other related models.

⁶This means that each article has been assigned a quality score within the dataset itself.

5.1.2 Readability

In GneissWeb, readability is defined as a function of sentence length, average number of words per sentence, number of syllables per word on average, and other associated text statistics; these serve as a proxy for algorithmically determining how coherent documents are, with higher scores denoting more difficult text. The aforementioned procedure produces a score, which can be compared to a threshold in order to separate high quality examples from low quality ones.

Observation 20. *Using one shared threshold for all data can be ineffective; different categories of text may have different readability score distributions, meaning that a single threshold can lead to certain categories being under or over-represented.*

Therefore, category-specific thresholds are applied in order to alleviate the aforementioned problem. Document categories are determined by category classifiers trained on specific data sources pertaining to specific key categories such that they each assign a "belonging" score for any given document to the category. Then, the document is assigned to the category that has the classifier producing the highest score.

5.1.3 Results

When applying these filters, as well as the others defined in [3], it was found that the initial dataset of 15T tokens was compressed to 10T tokens of "higher quality", leading to lower model training times and better downstream performance when applying the aforementioned models to language tasks such as reasoning and problem solving.

6 Defining ϕ_j

We can define ϕ_j , or any given selection function eventually composed into ϕ , as a combination of a utility function and selection mechanism, with the former indicating the "quality" of an article, as described in the previous section, and the latter indicating whether a given data point will be selected. These can be defined as follows:

Definition 21 (Utility Function). *The utility function $v(x^{(i)}) : D \rightarrow \mathbb{R}$ defines the overall quality of a given document, which can be a function of its readability or its score assigned by a classifier, among other potential metrics.*

Definition 22 (Selection Mechanism). *The selection mechanism is the method by which data points are selected; it uses the result of a point's utility function to determine whether it is of sufficient quality, and then decides both whether it should appear and how many times it should appear based on the result.*

From these definitions, it is apparent that GneissWeb defines utility functions when it assigns scores to documents, which are then subject to selection mechanisms to filter down the dataset. Most importantly, however, is that we can use these paradigms to define our goals for data selection, such as sampling around "good" data.

6.1 Distribution Matching

Definition 23 (Distribution Matching). *In distribution matching, selection mechanisms aim to select data points similar to those in a target distribution of data such that the resulting distribution is akin to the target one.*

We have previously defined the target distribution as that of "good" data sourced from websites such as Wikipedia and Arxiv, though it is important to note that it can relate to other aspects of data. Distribution matching can be achieved in two main ways, those being to alter specific data points or alter the dataset, with the former involving editing the content found within a document such that the resulting text is in-distribution, and the latter using the selection mechanism on the utility function across the data.

Definition 24 (Dataset Alteration). *We can define such a function, $\phi_j : D \rightarrow \mathbb{N}_0$, as mapping all data points $\in D$ using the utility function, then applying a selection threshold to decide whether the data point should be included, duplicated, or removed. The utility function and threshold are selected such that they act as a proxy for similarity to a target distribution.*

Part III

MinHash in Practice: Large-Scale Deduplication for LLM Pre-training (not covered in class; by *Xinyu Lu*)

7 From MinHash to Practice

In the lecture, documents are represented using sets of k -shingles, and Jaccard similarity is used to measure the amount of overlap between two documents. MinHash provides a compact way to estimate this similarity:

$$\Pr[h_\pi(A) = h_\pi(B)] = J(A, B).$$

[2]

Repeating MinHash with several independent hash functions gives each document a compact signature. However, MinHash alone does not solve the full deduplication problem. If there are n documents, comparing every pair still requires considering $O(n^2)$ pairs. At web scale, this is prohibitively expensive.

The practical question is therefore not only how to estimate similarity, but also how to efficiently find the small number of document pairs that are likely to be near duplicates. Locality-sensitive hashing (LSH) provides one solution to this problem [5].

8 LSH for Candidate Generation

Suppose each document has a MinHash signature containing r hash values. We divide the signature into b bands, with q hash values in each band, so that

$$r = bq.$$

Two documents become a candidate pair if their signatures are identical in at least one band. If their Jaccard similarity is s , the probability that all q hash values in one band agree is approximately

$$s^q.$$

Therefore, the probability that the two documents agree in at least one of the b bands is approximately

$$1 - (1 - s^q)^b.$$

This gives LSH a useful thresholding effect. Documents with high similarity are likely to become candidates, while documents with low similarity are unlikely to collide in any band. The expensive similarity computation can therefore be restricted to the candidate pairs.

The resulting pipeline is

documents $\rightarrow$ k -shingles $\rightarrow$ MinHash $\rightarrow$ LSH candidates $\rightarrow$ verification $\rightarrow$ deduplicated corpus.

The important distinction is that MinHash estimates similarity, whereas LSH makes those estimates searchable at large scale.

9 A Real Example: FineWeb

A practical example is FineWeb [9], a large-scale dataset constructed from Common Crawl for LLM pre-training. Its data-processing pipeline includes extensive filtering and deduplication, illustrating how similarity-based methods can be used on web-scale data.

FineWeb uses MinHash-based deduplication on text represented by n -grams. In its documented procedure, 5-grams are used together with multiple hash functions to identify documents with substantial textual overlap [4]. This allows repeated web content to be detected without explicitly comparing every possible pair of documents.

The choice of n -gram size is important. Individual words are often too common to provide a useful indication of duplication, while longer sequences capture more information about the actual wording of a document. Thus, the representation itself affects which documents are considered similar.

FineWeb also illustrates an important point about large-scale deduplication: MinHash is only one component of a larger data-processing pipeline. The system must choose an appropriate shingle size, construct compact signatures, generate candidate pairs, and finally apply a similarity criterion to determine which documents should be removed.

10 Practical Trade-offs

A deduplication system must balance computational efficiency with the risk of removing useful data. Two types of errors are possible: false positives, where distinct documents are incorrectly treated as duplicates, and false negatives, where duplicate documents are missed.

These errors depend on several parameters, including the shingle size, the number of MinHash functions, the LSH band configuration, and the similarity threshold. For example, a lower similarity threshold may identify more overlapping documents but can also remove documents that share common phrases without being true duplicates.

There is also an important distinction between deduplication and quality filtering. Deduplication removes repeated information, whereas quality filtering attempts to remove information that is unsuitable for training. A document can therefore be unique but low quality, or repetitive but still useful. In practice, these are complementary operations rather than substitutes.

At web scale, these design choices become particularly important because even a small error rate can affect a large number of documents. The practical performance of MinHash therefore depends not only on its theoretical similarity guarantee, but also on how the entire deduplication pipeline is designed.

11 Conclusion

The lecture introduces MinHash as a compact estimator of Jaccard similarity. LSH provides the next step needed for large-scale deduplication by turning MinHash signatures into an efficient candidate search mechanism.

The resulting idea is simple:

MinHash estimates similarity $\rightarrow$ LSH finds candidates $\rightarrow$ verification removes duplicates.

FineWeb demonstrates how this theoretical framework can be incorporated into a real LLM pre-training pipeline. The broader lesson is that an algorithm that is effective in theory must also be combined with appropriate representations, parameters, and filtering procedures to work reliably at web scale.

References

- [1] Alon Albalak, Yanai Elazar, Sang Michael Xie, Shayne Longpre, Nathan Lambert, Xinyi Wang, Niklas Muennighoff, Bairu Hou, Liangming Pan, Haewon Jeong, Colin Raffel, Shiyu Chang, Tatsunori Hashimoto, and William Yang Wang. A survey on data selection for language models, 2024.
- [2] Andrei Z. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences*, pages 21–29. IEEE, 1997.
- [3] Hajar Emami Gohari, Swanand Ravindra Kadhe, Syed Yousaf Shah, Constantin Adam, Abdulhamid Adebayo, Praneet Adusumilli, Farhan Ahmed, Nathalie Baracaldo Angel, Santosh Subhashrao Borse, Yuan-Chi Chang, Xuan-Hong Dang, Nirmal Desai, Revital Eres, Ran Iwamoto, Alexei Karve, Yan Koyfman, Wei-Han Lee, Changchang Liu, Boris Lublinsky, Takuyo Ohko, Pablo Pesce, Maroun Touma, Shiqiang Wang, Shalisha Witherspoon, Herbert Woisetschläger, David Wood, Kun-Lung Wu, Issei Yoshida, Syed Zawad, Petros Zerfos, Yi Zhou, and Bishwaranjan Bhattacharjee. Gneissweb: Preparing high quality data for llms at scale, 2025.
- [4] Hugging Face. Fineweb: a 15t token web dataset, 2024. FineWeb documentation.
- [5] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, pages 604–613. ACM, 1998.
- [6] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger, editors, *Automata, Languages and Programming*, pages 943–955, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.
- [7] Sedir Mohammed, Lukas Budach, Moritz Feuerpfel, Nina Ihde, Andrea Nathansen, Nele Noack, Hendrik Patzlaff, Felix Naumann, and Hazar Harmouch. The effects of data quality on machine learning performance on tabular data. *Information Systems*, 132:102549, July 2025.
- [8] Ge Nong, Sen Zhang, and Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. In *Proceedings of the 2009 Data Compression Conference, DCC '09*, page 193–202, USA, 2009. IEEE Computer Society.
- [9] Guilherme Penedo, Hynek Kydlíček, Loubna Ben Allal, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro Von Werra, and Thomas Wolf. The fineweb datasets: Decanting the web for the finest text data at scale, 2024.