

# Lecture 14: Data part 2

Camille Morencé and Yixuan Li

*Stanford CS336 — Spring 2025*

## 1 Main Goal of Filtering

Given some target data  $T$  and lots of raw data  $R$ , the goal is to find  $T' \subseteq R$  that resembles  $T$ . While  $T$  represents the target distribution on which we want the LLM to perform well, it is often small, expensive, or partially unavailable. We can think of  $T$  as a probability distribution  $p_T(x)$  which we want our LLM to estimate well. On the other hand, raw data  $R$  are often huge and cheap, but noisy, with distribution  $p_R(x) \neq p_T(x)$ . The goal of finding  $T' \subseteq R$  is to retrieve large domain-specific data resembling  $T$  on which we want our LLM to be trained and evaluated on, such that  $p_{T'}(x) \approx p_T(x)$ . To do so,  $R$  must be filtered from low-quality, noisy, and duplicate data. The following topics are going to be discussed in this report:

- Filtering algorithmic tools (KenLM, fastText, DSIR)
- Filtering applications (Language identification, Quality filtering, Toxicity filtering)
- Deduplication (Exact and Near duplicates)

## 2 Filtering algorithmic tools

The goal for all the filtering algorithms is to define a score for all examples  $x \in R$  to find if they are similar to  $T$ .

### 2.1 KenLM

#### 2.1.1 Classical n-gram model

KenLM is a fast implementation of a classical n-gram language model. In other words, given an item with a token sequence  $w_1, w_2, \dots, w_T$ , their joint probability is factorized as follow:

$$P(w_1, \dots, w_T) = \prod_{t=1}^T P(w_t | w_1, \dots, w_{t-1}).$$

for all tokens appearing before. However, an n-gram approximation assumes a finite context:

$$P(w_t | w_1, \dots, w_{t-1}) \approx P(w_t | w_{t-n+1}, \dots, w_{t-1})$$

Let's define  $h_t = (w_{t-n+1}, \dots, w_{t-1})$  be an (n-1)-gram appearing before  $w_t$ , then we can rewrite the above equation as:

$$P(w_1, \dots, w_T) \approx \prod_{t=1}^T P(w_t | h_t)$$

To optimize this, we can use the MLE. First, let  $C(h, w)$  be the count of the n-gram  $(h, w)$  (ex: "cat in the hat") and  $C(h)$  be the count of its history (ex: "car in the"), then the MLE estimate is:

$$P_{\text{MLE}}(w | h) = \frac{C(h, w)}{C(h)}$$

and the log-likelihood of an item  $i$  is:

$$\log P(i) = \sum_{t=1}^T \log P(w_t | h_t)$$

However, the problem with this method is if  $C(h, w) = 0$ , then  $P_{\text{MLE}}(w | h) = 0$ . In other words, if our model hasn't encountered a specific sequence of tokens (ex: never seen "durian" after "I eat a"), then it will put the probability of generating a token after a given history down to 0, which prevents the generation of rare sequences.

### 2.1.2 Backoff Model

A general backoff formulation is

$$P(w | h) = \begin{cases} P_{\text{high}}(w | h) & \text{if } C(h, w) > 0, \\ \alpha(h)P(w | h') & \text{otherwise,} \end{cases}$$

where

$$h' = (w_{t-n+2}, \dots, w_{t-1}),$$

is the (n-2)-gram in the history of the (n-1)-gram finishing with token  $w$ , and  $\alpha(h)$  redistributes leftover probability mass.

### 2.1.3 Modified Kneser–Ney Smoothing

KenLM typically implements Modified Kneser–Ney (MKN) smoothing.

Instead of raw counts, we subtract a discount  $D \in (0, 1)$ :

$$P_n(w | h) = \frac{\max(C(h, w) - D, 0)}{C(h)} + \lambda(h)P_{n-1}(w | h').$$

The backoff weight is

$$\lambda(h) = \frac{D \cdot N_1^+(h, \cdot)}{C(h)},$$

where

$$N_1^+(h, \cdot) = |\{w : C(h, w) > 0\}|.$$

counts the number of distinct contexts in which  $h$  appears.

and with base case (unigram):

$$P_1(w) = \frac{N_1^+(\cdot, w)}{\sum_{w'} N_1^+(\cdot, w')}.$$

where

$$N_1^+(\cdot, w) = |\{h : C(h, w) > 0\}|$$

counts the number of distinct contexts in which  $w$  appears.

Moreover, KenLM stores probabilities in log-space. The item score is

$$\text{score}(i) = \sum_{t=1}^T \log P_n(w_t | h_t).$$

A length-normalized score is

$$\frac{1}{T} \sum_{t=1}^T \log P_n(w_t | h_t).$$

Remainder, perplexity is defined as

$$\text{PPL}(i) = \exp\left(-\frac{1}{T} \sum_{t=1}^T \log P_n(w_t | h_t)\right).$$

Lower perplexity indicates that the model assigns a higher probability to the sequence.  
Training minimizes empirical cross-entropy:

$$\mathcal{L} = -\mathbb{E}_{x \sim \hat{p}_{\text{data}}} \log p_{\text{model}}(x).$$

Thus the  $n$ -gram model approximates

$$p_{\text{model}}(x) \approx p_T(x)$$

under a finite-order Markov assumption and we keep  $x$  if  $\text{score}(x) \geq \text{threshold}$ .

## 2.2 fastText: A Fast Linear Text Classifier

**Goal.** Given labeled text data

$$\{(x^{(t)}, y^{(t)})\}_{t=1}^N, \quad y^{(t)} \in \{1, \dots, K\},$$

learn a classifier that is:

- computationally extremely fast,
- scalable to very large vocabularies,
- suitable for large-scale data filtering.

**Text Representation.** Let the vocabulary size be  $V$ . Each token in the vocabulary is assigned a unique integer in  $\{1, \dots, V\}$ .

A document of length  $L$  is represented as a sequence of token indices:

$$x = (x_1, x_2, \dots, x_L), \quad x_i \in \{1, \dots, V\}.$$

Here:

- $L$  = number of tokens in the document,
- $V$  = vocabulary size,
- $x_i$  = index of the  $i$ -th token.

**Embedding Layer.** Each token index  $x_i$  is mapped to an embedding vector:

$$W \in \mathbb{R}^{V \times H},$$

where:

$$W_{x_i} \in \mathbb{R}^H$$

is the embedding of token  $x_i$ .

The hyperparameter

$$H = \text{embedding dimension}$$

controls model capacity. Typically  $H \ll V$  and  $H \ll K$ .

**Document Representation (Mean Pooling).** fastText uses a bag-of-words representation. The document embedding is the average:

$$\bar{h}(x) = \frac{1}{L} \sum_{i=1}^L W_{x_i} \in \mathbb{R}^H.$$

Order of tokens does not matter.

**Linear Classification Head.** Define

$$U \in \mathbb{R}^{H \times K}.$$

Logits:

$$z(x) = U^\top \bar{h}(x) \in \mathbb{R}^K.$$

Prediction:

$$\hat{p}(y | x) = \text{softmax}(z(x)).$$

**Training Objective.** Given training data  $\{(x^{(t)}, y^{(t)})\}_{t=1}^N$ , optimize cross-entropy:

$$\mathcal{L}(W, U) = - \sum_{t=1}^N \log \hat{p}(y^{(t)} | x^{(t)}).$$

Parameters updated by SGD:

$$(W, U) \leftarrow (W, U) - \eta \nabla \mathcal{L}.$$

**$n$ -grams via Hashing Trick.** To incorporate bigrams or trigrams, we treat each  $n$ -gram as an additional token.

Since the number of possible  $n$ -grams is unbounded, we hash each  $n$ -gram string  $g$  into one of  $B$  bins:

$$\phi(g) = (\text{hash}(g) \bmod B) + 1.$$

These hashed indices are appended to the vocabulary.

Thus the effective vocabulary size becomes:

$$V_{\text{total}} = V_{\text{tokens}} + B.$$

Then:

$$x_i \in \{1, \dots, V_{\text{total}}\}.$$

---

**Algorithm 1** fastText with hashed  $n$ -grams (end-to-end, single-sample SGD form)

---

**Require:** Token vocabulary size  $V_{\text{tok}}$ ; hash bins  $B$ ; embedding dim  $H$ ; classes  $K$ ;  $n$ -gram orders  $\mathcal{N}$  (e.g.  $\{2, 3\}$ ); learning-rate schedule  $\{\eta_s\}$ .

1:  $V_{\text{total}} \leftarrow V_{\text{tok}} + B$   
2: Initialize parameters:  $W \in \mathbb{R}^{V_{\text{total}} \times H}$  (embeddings),  $U \in \mathbb{R}^{H \times K}$  (linear head)  
3: **for** training step  $s = 1, 2, \dots$  **do**  
4:   Sample one training pair  $(x, y)$ , where  $x = (x_1, \dots, x_L)$  and  $y \in \{1, \dots, K\}$   
5:   **Build feature index multiset**  $F(x)$ :  
6:    $F \leftarrow [x_1, \dots, x_L]$  (token indices, each  $x_i \in \{1, \dots, V_{\text{tok}}\}$ )  
7:   **for** each  $n \in \mathcal{N}$  **do**  
8:     **for** each  $n$ -gram position  $t = 1, \dots, L - n + 1$  **do**  
9:       Form  $n$ -gram string  $g \leftarrow (x_t, \dots, x_{t+n-1})$  (token tuple)  
10:        $b \leftarrow (\text{hash}(g) \bmod B) + 1$  ( $b \in \{1, \dots, B\}$ )  
11:       Append hashed index  $V_{\text{tok}} + b$  to  $F$   
12:     **end for**  
13:   **end for**  
14:    $M \leftarrow |F|$   
15:   **Forward pass:**  
16:    $\bar{h} \leftarrow \frac{1}{M} \sum_{j=1}^M W_{F_j}$  ( $\bar{h} \in \mathbb{R}^H$ )  
17:    $z \leftarrow U^\top \bar{h}$  ( $z \in \mathbb{R}^K$ )  
18:    $p_k \leftarrow \frac{\exp(z_k)}{\sum_{k'=1}^K \exp(z_{k'})}$  for  $k = 1, \dots, K$   
19:   **Loss:**  $\ell \leftarrow -\log p_y$   
20:   **Backward pass (SGD update):**  
21:    $g_z \leftarrow p - e_y$  (where  $(e_y)_k = \mathbb{I}[k = y]$ )  
22:    $g_U \leftarrow \bar{h} g_z^\top$  ( $H \times K$ )  
23:    $g_{\bar{h}} \leftarrow U g_z$  ( $H$ -vector)  
24:   For each index  $i$  appearing in  $F$  with multiplicity  $c_i$ :  $g_{W_i} \leftarrow \frac{c_i}{M} g_{\bar{h}}$   
25:    $U \leftarrow U - \eta_s g_U$   
26:   For each index  $i$  updated above:  $W_i \leftarrow W_i - \eta_s g_{W_i}$   
27: **end for**  
28: **return**  $W, U$

---

## 2.3 DSIR

Density-Ratio Based Data Selection (DSIR) is an explicitly statistical method whose core idea is to find data in  $R$  that matches the target distribution  $T$ . Formally, we want samples  $x \sim R$  that are likely under  $p_T$ .

Therefore, we can define the density ratio (which is used here as the score):

$$r(x) = \frac{p_T(x)}{p_R(x)}.$$

where

- $r(x) > 1$  means  $x$  is more likely under the target distribution.
- $r(x) < 1$  means  $x$  is more typical of the raw distribution.

Selecting top- $k$  samples by  $r(x)$  gives a subset closest to  $T$ .

If we want expectations under  $p_T$  but only have samples from  $p_R$ :

$$\mathbb{E}_{x \sim p_T}[f(x)] = \mathbb{E}_{x \sim p_R}[r(x)f(x)].$$

Thus  $r(x)$  serves as an importance weight.

### 3 Filtering applications

Once you have a filtering model (KenLM / fastText / any classifier), you can reuse the same pipeline for different "what to keep" goals:

- Language identification (keep English)
- Quality filtering (keep "high-quality" webpages)
- Toxicity filtering (remove hate/NSFW/toxic text)

General pattern:

$$\text{score}(x) \Rightarrow \text{keep } x \text{ if } \text{score}(x) \geq \tau \text{ (often stochastically)}$$

#### 3.1 Language identification

The goal of Language identification is identify text of a specific language within our data. High-quality filtering, deduplication, toxicity removal, formatting cleanup, etc, are usually language-specific. Doing this well for 100+ languages is much harder than for one. Moreover, if training tokens are split across many languages, then each language get less tokens, making per-language performance degrade.

##### 3.1.1 FastText for Language Identification

FastText for Language Identification is a lightweight text classification model from Meta AI meant to classify text according to their language. It supports 176 languages and was trained on Wikipedia, Tatoeba (translation site) and SETimes (Southeast European news). After being input a prompt, it returns a language label (ex, en for English) as well as a confidence level (ex, 0.92 for 92% confidence). Dolma is a three-trillion-token English corpus, built from a diverse mixture of web content, scientific papers, code, public-domain books, social media, and encyclopedic materials which used FastText for Language Identification and kept pages that output at least a confidence level of 50% in English.

Notable limitations of this method are:

- Often misclassifies latex since it lacks natural-language cues. Often is classified as English or "unknown".
- Code snippets are also often classified as English since training data lacked programming language examples.

- Short sequences are difficult to identify because there is insufficient signal.
- Ill-defined problem for code switching (ex, sentences that use multiple languages) since it forces a single label.
- Dialects can also be misclassified (ex, Scottish English may be misclassified non-English).
- Similar languages like Malay vs Indonesian are hard to classify.

### 3.1.2 OpenMathText

OpenMathText is an open dataset of high-quality mathematical web text whose goal was to extract high-quality mathematical text from Common Crawl. Therefore, instead of filtering by language, they filtered by domain (mathematics). The pipeline is as follow:

- Rule-based filtering: Only keep documents containing LaTeX commands, Math symbols, and Theorem-like structures.
- Perplexity filtering: Train a KenLM on ProofPile (math corpus). Then, compute the perplexity of candidate documents and keep only those with a perplexity less than 15000.
- fastText classifier: Train classifier such that it's positive for math text and negative for non-math text. Their threshold was above 0.17 for math and above 0.8 for non-math. These asymmetric thresholds are meant to balance recall vs precision. 0.17 is meant to reduce false negatives since they don't accidentally throw away real math and the perplexity filtering already throws away lots of noise.

In the end, they produce  $14.7B$  math tokens and trained  $1.4B$  parameter models, outperforming models trained on  $20\times$  more general data.

## 3.2 Quality filtering

The goal of quality filtering is to select training data that improves downstream model performance (reasoning, coding, factuality, etc.) while reducing noise from raw web crawls like CommonCrawl. Quality filtering is now the norm, and it trains a classifier to distinguish “high-quality” text from general web noise.

### 3.2.1 GPT-3 Filtering

This model classifies data from Wikipedia, WebText2, Books1 and Books2 as positives (high-quality) while classifying as negative random sample from CommonCrawl. Their method is as follows:

- Train a linear classifier on bag-of-words features.
- Score each CommonCrawl document.
- Keep documents stochastically based on score: `"np.random.pareto(9) < 1 - score"`. Instead of a hard cutoff, this is meant to keep diversity, avoid collapsing to only Wikipedia-style text, and sample proportionally to “quality.”

This method allows GPT3 to not be overly aggressive in its filtering, which can otherwise reduce domain diversity. In comparison, LLaMA and RedPajama Filtering are using a hard cutoff, which improves average quality but may reduce stylistic diversity.

### 3.2.2 Phi-1

Phi-1 has for its core idea to train a small model ( $1.3B$ – $1.5B$ ) on extremely high-quality, educational data instead of scaling data size. Their pipeline is as follows:

- Given raw data  $R$  from The Stack, use GPT-4 to classify 100K samples using the prompt: "determine its educational value for a student whose goal is to learn basic coding concepts". This produces positive examples of target data  $T$
- Train classical classifier: for each snippet of code, they pass it through a pretrained codegen model which allows for deep semantic understanding of code compare to  $n$ -gram. With this, they extract the embedding.
- Train a random forest classifier on these embeddings and select only examples classified as positive.

Training of this filtered subset has allowed for better performance and faster convergence (12.19% after 96K steps compared to 17.68% after 36K steps)

## 3.3 Toxicity filtering

Goal: Remove toxic, hateful, obscene, or unsafe content from large web-scale corpora before language model training.

### 3.3.1 Dataset Used: Jigsaw Toxic Comments (2018)

- Source: Wikipedia talk page comments
- Goal: improve online discussions
- Human-annotated labels (toxic, severe\_toxic, obscene, threat, insult, identity\_hate)

Thus, we have supervised data:

$$(x^{(t)}, y^{(t)})$$

where  $x^{(t)}$  = comment text, and  $y^{(t)}$  = toxicity-related label(s).

### 3.3.2 Model Used: fastText Classifiers

Dolma trains two separate fastText models:

A Hate classifier

- Positive class = {unlabeled, obscene} }
- Negative class = all other categories
- Purpose: detect hate-related language.

B NSFW classifier

- Positive class = {obscene} }
- Negative class = all others
- Purpose: detect sexual / explicit content.

### 3.3.3 Model Architecture

Same fastText structure discussed earlier: Document representation

$$\bar{h}(x) = \frac{1}{L} \sum_{i=1}^L W_{x_i}$$

Classification

$$z = U^\top \bar{h}(x)$$
$$\hat{p}(y | x) = \text{softmax}(z)$$

Binary case:

$$K = 2$$

So:

$$\hat{p}(\text{ toxic} | x)$$

is the toxicity score.

Typical filtering rule: Remove  $x$  if  $\hat{p}(\text{ toxic} | x) \geq \tau$  or equivalently: Keep  $x$  if  $\hat{p}(\text{ toxic} | x) < \tau$

Threshold  $\tau$  is tuned based on:

- desired safety level
- false positive tolerance
- recall vs precision tradeoff

## 4 Deduplication

Deduplication's main goal is to detect exact duplicates (straightforward) or near duplicates (very similar text that differs by only a few tokens) within the raw data  $R$  and retain either none, only one, or a few of these. This step enhances training by preventing overfitting on repeated data, making training more efficient by using fewer tokens, and avoiding memorization. Challenges arise from this step. Since it is fundamentally about comparing items to other items, this step can become costly and time-consuming, which forces the need for linear-time algorithms to scale. In this section, we discuss how to deal with both exact duplicates and near duplicates.

### 4.1 Exact duplicates

As the name implies, exact duplicates happen when two subsets of the raw data are perfectly identical (same tokens, same size, same order, etc), which often implies that we want to keep at most one of them. Deleting for exact duplicates is often done early in the filtering process. However, token-by-token comparison is time-heavy ( $O(n)$  per comparison), so the main solution is to use a hash-based comparison.

#### 4.1.1 Hash-based comparison

Hash function  $h$  maps an item to a hash value, which considerably reduces its size. The goal is to find hash collisions, i.e., cases where  $h(x) = h(y)$   $x \neq y$ . In this case, if a collision occurs, it means that  $x$  and  $y$  were mapped to the same hash value, which is only possible if they are identical items. In this case, each comparison can be checked in constant time. However, this method can be cost-heavy as memory grows with the number of items stored.

#### 4.1.2 Bloom filter

A Bloom filter is a memory-efficient probabilistic data structure for checking whether an element is in a set. This method uses a bitarray instead of storing full items. The basic framework is as follows:

1. You have a bit array of size  $m$  (number of bins). All bits start at 0.
2. Each item is passed through a hash function, which gives an index into the bit array.
3. That bit is set to 1.
4. To query an item, hash it to an index. If 1, then maybe duplicate; if 0, not a duplicate.

A small number of bins leads to a higher chance in collision for non-items (something not actually in the set being tracked with the Bloom filter). A false positive might arise because all the bits a non-item is hashed to happen to be set by other items. Possible solutions to that are:

- increase the number of bins  $m$  (Naive fix since it decreases the error probability polynomially at the cost of memory usage)
- Use multiple hash functions  $k$  (Better fix, since it decreases the error probability exponentially while better balancing memory, computation, and accuracy.)

In the better fix, each item passes through multiple hash functions, and to have a collision, all indices returned by all the multiple hash functions must be mapped to 1.

#### 4.1.3 False Positive rate

Let

- $m$  = number of bits in the Bloom filter,
- $k$  = number of hash functions,
- $n$  = number of items inserted.

where each item sets  $k$  bits in the bit array and a query for a non-item also hashes to  $k$  bits.

For a single bit  $i$ :

$$P(\text{bit } i = 0 \text{ after inserting 1 item}) = 1 - \frac{1}{m},$$
$$P(\text{bit } i = 1 \text{ after inserting 1 item}) = \frac{1}{m}.$$

For one item hashed  $k$  times:

$$P(\text{bit } i = 0) = \left(1 - \frac{1}{m}\right)^k,$$

$$P(\text{bit } i = 1) = 1 - \left(1 - \frac{1}{m}\right)^k.$$

Probability that a given bit is still 0 after inserting  $n$  items:

$$P(\text{bit } i = 0) = \left(1 - \frac{1}{m}\right)^{kn}$$

Probability that the bit is 1:

$$f = 1 - \left(1 - \frac{1}{m}\right)^{kn}$$

A false positive occurs only if *all k bits for the query* are 1. Assuming independence:

$$P(\text{false positive}) = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k$$

Note that if  $k$  is large, then  $(1 - \frac{1}{m})^{kn} \approx e^{-kn/m}$ . Then, for fixed  $m$  and  $n$ , the optimal number of hash functions is:

$$k_{\text{opt}} = \frac{m}{n} \ln 2$$

The resulting false positive rate:

$$f \approx 0.5^k$$

In other words, if we happen to have  $k_{\text{opt}} = \ln 2 * m/n$ , then the probability of any given bit being 1 is  $\approx 0.5$ . Key notes of bloom filters here are:

- Bits may be set by multiple different items.
- Fixed memory usage, but no deletion (once a bit is set, it cannot be safely deleted).
- No false negatives, but small chances of false positives.

#### 4.1.4 C4 Example

C4 stands for Colossal Clean Crawled Corpus and is a massive dataset created from Common Crawl (web pages scraped from the internet). Its goal is to build a very large, high-quality English text corpus for pretraining the T5 model with hundreds of gigabytes of text (after filtering, the raw crawl is trillions of tokens).

C4 applies multiple filters such as keep only English text (language detection filter), remove boilerplate, HTML, JavaScript, and non-text content and remove near-duplicates, including 3-sentence spans. A “3-sentence span” is just three consecutive sentences in a document. Therefore, if a span appears more than once anywhere in C4, they keep only one copy. Exact match is used so the text has to match character-for-character.

The reason to use a “3-sentence span” is as follow:

- Single sentences are too common, and removing them could delete too much normal content.
- Longer spans (5–10 sentences) are rarer, so 3 is a practical compromise.
- Note: removing spans from the middle of a document can make it less coherent, but overall, this is better than keeping massive duplicated content.

## 4.2 Near Duplicates

Near duplicates are almost identical items (similar tokens, similar size, etc) which are redundant for the LLM to be trained on (ex: "Machine learning is great" and "Machine learning is really great"). The challenge here is both to define them and to detect them, since we cannot use hash or bit collisions.

### 4.2.1 Shingling

A shingle is a sequence of tokens (words or characters) of a specific size. For example, defining shingling of size 3 from "machine learning is great" would give 2 shingling, "machine learning is", and "learning is great". Each item becomes a set of shingles, and the key idea is that similar items share many shingles. Moreover, the use of shingling allows for less rigidity on the order of the sequences of tokens in a document. For instance, similar items can be made of the same tokens, but rearranged differently, which exact duplicates methods cannot detect.

### 4.2.2 Similarity measure (Jaccard similarity)

A common similarity metric is the Jaccard similarity:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

where  $A$  and  $B$  are sets of elements (ex: shingles of documents).

- $J(A, B) = 1.0 \rightarrow$  identical sets
- $J(A, B) = 0.8 \rightarrow$  very similar sets
- $J(A, B) = 0.0 \rightarrow$  totally different sets

Two documents are said to be near duplicates if their Jaccard similarity  $\geq$  a threshold.

However, Jaccard similarity for all pairs of documents is  $\mathcal{O}(n^2)$ , which can become infeasible for large datasets. Therefore, we need a way to compute this similarity in a linear time.

### 4.2.3 MinHash

MinHash provides a way to compress large sets of shingles into small signature vectors while preserving approximate similarity. The goal of MinHash is to have a random hash function  $h$  so that  $p[h(A) = h(B)] = Jaccard(A, B)$ . The core idea is the following

1. Let each document be represented as a set of shingles:  $S = \{s_1, s_2, \dots, s_m\}$ .
2. Apply  $k$  independent hash functions  $h_1, h_2, \dots, h_k$  to each shingle in the set.
3. For each hash function  $h_i$ , record the *minimum hash value* over all shingles in the set:

$$\text{MinHash}_i(S) = \min\{h_i(s) : s \in S\}$$

4. Concatenate these minimum values to form the MinHash signature vector:

$$\text{Signature}(S) = [\text{MinHash}_1(S), \text{MinHash}_2(S), \dots, \text{MinHash}_k(S)]$$

For two sets  $A$  and  $B$ , the probability that their MinHash values match for a given hash function  $h_i$  is exactly equal to their Jaccard similarity:

$$\Pr [\text{MinHash}_i(A) = \text{MinHash}_i(B)] = J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Proof: Let  $A$  and  $B$  be two sets, and define their union  $U = A \cup B$ . Let  $\pi$  be a random permutation of  $U$  (hash), and let  $\min_{\pi}(S)$  denote the element of  $S$  appearing first (min) under  $\pi$ . Then

$$\min_{\pi}(A) = \min_{\pi}(B) \iff \text{min element in } U \text{ under } \pi \text{ is in } A \cap B.$$

Since the first element of  $U$  is chosen uniformly at random,

$$\Pr[\text{MinHash}_i(A) = \text{MinHash}_i(B)] = \Pr[\min_{\pi}(A) = \min_{\pi}(B)] = \frac{|A \cap B|}{|U|} = J(A, B).$$

Hence, comparing these small signature vectors allows us to efficiently estimate Jaccard similarity without comparing the full sets, and in linear time.

#### 4.2.4 Locality sensitive hashing (LSH)

After computing MinHash signatures, comparing every pair of items is still computationally expensive for large datasets. Locality-Sensitive Hashing is a technique that allows us to efficiently identify likely near-duplicate items without performing all pairwise comparisons. The main idea of LSH is:

- Given  $k$  hash functions, Split the MinHash signature of each document into  $b$  bands, each containing  $r$  rows. Example for  $k = 12$ ,  $b = 3$  and  $r = 4$ :

$$h_1 h_2 h_3 h_4 | h_5 h_6 h_7 h_8 | h_9 h_{10} h_{11} h_{12}$$

where  $k = b * r$ .

- Hash each band independently on  $r$  hash table.
- Two items are considered a candidate pair if they hash to the same bucket in any band.

Therefore, if two items have identical hash values for all hash functions within a single band, LSH considers them likely near-duplicates and marks them as candidate pairs for further verification. Conversely, if two items do not match in any band, LSH considers them unlikely to be duplicates, and no further comparison is needed for that pair. Importantly, the goal of LSH is not to definitively detect near-duplicates, but to efficiently generate candidate pairs, thereby accelerating deduplication by avoiding comparisons of all possible pairs.

There are some trade-offs to consider when setting each parameter:

- Number of rows per band  $r$ :** Larger  $r$  makes matches within a band stricter (fewer false positives), because all  $r$  hash values in the band must match exactly.
- Number of bands  $b$ :** More bands increase the chance that near-duplicate documents are detected in at least one band (reduces false negatives), but can also slightly increase false positives.

Let  $sim$  be the similarity (Jaccard similarity) between two documents. The probability that a pair becomes a candidate in LSH is:

$$p_{\text{candidate}} = 1 - (1 - sim^r)^b$$

- High similarity  $s \approx 1 \rightarrow p_{\text{candidate}} \approx 1$  (almost always detected)
- Low similarity  $s \approx 0 \rightarrow p_{\text{candidate}} \approx 0$  (unlikely to be flagged)

The final step is to compute true similarity (Jaccard, cosine, edit distance, etc.) for each candidate pair and keep only one if these similarity scores are above a set threshold.