# Comp 480/580 - Assignment #4

Dev Sanghvi - `ds221`

Rice University
Date: 11/26/2025

## Problem Overview

This assignment implements a MinHash-based locality-sensitive hashing (LSH) search engine for near-duplicate URL retrieval on the AOL click log. Each URL is represented by character 3-grams, compressed into a fixed-length MinHash signature, and indexed into multiple hash tables via LSH. On top of this engine, I empirically compare LSH lookup time and quality against a brute-force Jaccard baseline, sweep the $(K, L)$ parameters, and study the theoretical "S-curves" of retrieval probability $P_x = 1 - (1 - J_x^K)^L$.

The second part of the assignment analyzes an adaptive sampling estimator for a sum $T = \sum_i w_i$. Sampling indices $i$ with non-uniform probabilities $p_i$, I show that the importance-sampling estimator remains unbiased, derive its variance in terms of the $\{w_i\}$ and $\{p_i\}$, and discuss how to choose $p_i$ to outperform uniform sampling.

## 1 Implementation Summary

**Code layout.** The implementation is organized into a single Python module `MinHash.py`:[1]

- **3-gram representation.** A helper `get_kgrams(s, k=3)` lowercases each URL and returns the set of overlapping character 3-grams. A second helper computes Jaccard similarity between two sets of 3-grams.
- **MinHash signatures.** `MinHash(A, k)` takes a string $A$ and produces a signature of length $k$ by simulating $k$ independent hash functions. The $i$-th hash function is implemented by a fast deterministic routine `fast_minhash_code(i, shingle)`, which first computes a 32-bit polynomial hash of the shingle and then mixes in the integer seed $i$ using a Murmur-style 32-bit mixing sequence. The signature coordinate is the minimum of these codes over all shingles of $A$.
- **LSH hash tables.** `HashTable(K, L, B, R)` implements a banding-based LSH index on MinHash signatures of length $K \cdot L$:
  - Each of the $L$ tables corresponds to one contiguous band of $K$ MinHash coordinates; table $t$ uses coordinates $[tK, \ldots, tK + K - 1]$.
  - To insert an item, I slice its $K \cdot L$-dimensional signature into $L$ bands, form the key (table id, band values) for each band, map this key into the range $\{0, \ldots, R-1\}$ via the deterministic SHA-256–based routine `stable_hash`, and then fold that code into one of $B$ physical buckets by taking it modulo $B$. In all AOL experiments I set $R = 2^{20}$ and $B = 64$, matching the constants in the assignment handout.

[1] See the submitted `MinHash.py` for complete code and inline documentation.

  - To look up a query, I repeat the same banding and bucket computation and return the union of candidate ids across all tables.

- **AOL preprocessing.** `load_aol_urls()` reads the AOL log, extracts the `ClickURL` column, drops missing entries, and keeps the unique URLs. A helper `precompute_shingles_and_signatures()` builds a cached list of 3-gram sets and 600-dimensional MinHash signatures for all URLs; each LSH configuration with parameters $(K, L)$ uses only the first $K \cdot L$ coordinates of these precomputed signatures.

- **Experiment drivers.** A set of small functions drives each task:

  - `evaluate_lsh()` samples a fixed query set, runs LSH lookup for each query, computes Jaccard similarities with all retrieved candidates, and reports the mean Jaccard of (i) all candidates and (ii) the top-10 by Jaccard.
  - `brute_force_queries()` computes true Jaccard similarities between each query and every URL in the collection and measures the total wall-clock time.
  - `run_lsh_experiment_grid()` sweeps $K \in \{2, 3, 4, 5, 6\}$ and $L \in \{20, 50, 100\}$, reindexing for each $(K, L)$ pair and logging the mean Jaccard and query time.
  - `plot_s_curves()` generates the theoretical S-curve plots required in Task 4, saving them as PNGs.

**Verification on small strings.** For Task 0, I validate MinHash against the exact 3-gram Jaccard similarity on the two provided sentences $S_1$ and $S_2$. In my run, the true Jaccard similarity was

$$\mathrm{Jac}(S_1, S_2) \approx 0.5194,$$

and the empirical fraction of matching coordinates between two 100-dimensional MinHash signatures was

$$\widehat{\mathrm{Jac}}_{\mathrm{MinHash}} \approx 0.45.$$

These values differ by a modest sampling error consistent with using only 100 hash functions, and they confirm that the signature and LSH machinery are wired correctly.

## 2 Run Configuration

The only source of randomness in my implementation is the sampling of query URLs; MinHash signatures and LSH bucket mapping are deterministic functions of the data. MinHash signatures use the deterministic `fast_minhash_code` described above, while LSH buckets use the SHA-256–based `stable_hash`; both avoid Python's process-randomized `hash` and make the experiments repeatable.

Unless otherwise stated, all experiments use:

- **3-gram representation:** lowercased character 3-grams.

- **Precomputed MinHash signatures:** 600 coordinates per URL. For any LSH configuration $(K, L)$, I use only the first $K \cdot L$ coordinates; in particular, the baseline $(K, L) = (2, 50)$ uses $K \cdot L = 100$ MinHash values.

- **LSH parameters:** baseline configuration $(K, L, B, R) = (2, 50, 64, 2^{20})$, and the same $(B, R)$ pair for all points in the $(K, L)$ grid. Each band key is hashed deterministically into $\{0, \ldots, R-1\}$ and then folded into one of the $B$ physical buckets.

- **Queries:** 200 URLs sampled uniformly without replacement.

- **Dataset:** AOL query log file `user-ct-test-collection-01.txt` supplied with the assignment.

After running `MinHash.py` once, I record the key corpus statistics in Table 1.

| Metric | Value |
|---|---|
| Processed URLs | 377,870 |
| Unique URLs | 377,870 |
| Precomputed MinHash signature length | 600 |
| Baseline LSH MinHash coordinates $K \cdot L$ | 100 (with $K = 2$, $L = 50$) |
| Baseline LSH parameters $(K, L, B, R)$ | $(2, 50, 64, 2^{20})$ |
| Query count | 200 |
| Dataset path | `user-ct-test-collection-01.txt` |

Table 1: Run summary from the latest execution of `MinHash.py`.

# 3 LSH Retrieval Quality (Tasks 1 & 3)

## 3.1 Baseline: K=2, L=50 (Task 1)

For each of the 200 sampled query URLs, I:

1. use its precomputed 3-gram set and truncate its 600-dimensional MinHash signature to the first $K \cdot L = 100$ coordinates (for $K = 2$, $L = 50$),
2. retrieve candidate ids from the LSH index via `HashTable.lookup()`,
3. remove the query id from the candidate set (if present),
4. compute the true 3-gram Jaccard similarity with each candidate,
5. sort candidates by Jaccard and keep the top-10.

Let $\mathcal{C}_q$ denote all candidates returned for query $q$, and $\mathcal{T}_q$ the top-10 candidates by Jaccard. I report two aggregate quality metrics:

$$\overline{J}_{\text{all}} = \frac{1}{\sum_q |\mathcal{C}_q|} \sum_q \sum_{u \in \mathcal{C}_q} \text{Jac}(q, u),$$

$$\overline{J}_{\text{top-10}} = \frac{1}{10|\mathcal{Q}|} \sum_q \sum_{u \in \mathcal{T}_q} \text{Jac}(q, u),$$

where $\mathcal{Q}$ is the set of queries.

Table 2 records the mean Jaccard similarities and the measured query-time cost for this baseline configuration.

| Metric | Value |
|---|---|
| Mean Jaccard over all candidates $\overline{J}_{\text{all}}$ | 0.2273 |
| Mean Jaccard over top-10 candidates $\overline{J}_{\text{top-10}}$ | 0.5592 |
| Average LSH query time (seconds/query) | 0.5867 |

Table 2: Baseline LSH retrieval quality and query time for K=2, L=50.

Qualitatively, the LSH engine returns a small pool of candidates with noticeably higher Jaccard similarity than random URLs, and restricting to the top-10 per query further concentrates on strong near-duplicates.

### 3.2 Effect of $K$ and $L$ (Task 3)

To study the LSH parameter trade-offs, I sweep $K \in \{2, 3, 4, 5, 6\}$ and $L \in \{20, 50, 100\}$ with the bucket range fixed at $R = 2^{20}$ and $B = 64$ buckets per table. For each $(K, L)$, I rebuild the index, run the same 200 queries, and measure:

- mean Jaccard over all retrieved candidates,
- mean Jaccard over the top-10,
- average query time in seconds.

| $K$ | $L$ | Mean Jaccard (all) | Mean Jaccard (top-10) | Time/query (s) |
|---|---|---|---|---|
| 2 | 20 | 0.2381 | 0.5591 | 0.396479 |
| 2 | 50 | 0.2273 | 0.5592 | 0.556519 |
| 2 | 100 | 0.2186 | 0.5592 | 0.673724 |
| 3 | 20 | 0.2390 | 0.5580 | 0.249879 |
| 3 | 50 | 0.2297 | 0.5591 | 0.425635 |
| 3 | 100 | 0.2215 | 0.5592 | 0.584296 |
| 4 | 20 | 0.2301 | 0.5548 | 0.216542 |
| 4 | 50 | 0.2227 | 0.5587 | 0.392571 |
| 4 | 100 | 0.2202 | 0.5591 | 0.571201 |
| 5 | 20 | 0.2214 | 0.5489 | 0.235495 |
| 5 | 50 | 0.2200 | 0.5573 | 0.404181 |
| 5 | 100 | 0.2184 | 0.5590 | 0.573508 |
| 6 | 20 | 0.2177 | 0.5412 | 0.199269 |
| 6 | 50 | 0.2174 | 0.5546 | 0.380196 |
| 6 | 100 | 0.2176 | 0.5583 | 0.540356 |

Table 3: Grid search over (K,L): mean Jaccard over all candidates, mean Jaccard over the top-10, and mean query time. Values are from the observed run on the AOL dataset.

We see the expected trade-off pattern. As $K$ increases for a fixed $L$, the bands become more selective, so candidate sets shrink and the mean Jaccard over all candidates changes only mildly (staying in the $\approx$ 0.22–0.24 range), while the top-10 mean Jaccard remains tightly clustered around 0.55. For the most selective setting $(K, L) = (6, 20)$ the top-10 mean Jaccard drops to about 0.54, indicating that some true near-duplicates are now missed. Increasing $L$ at fixed $K$ increases the number of tables and therefore the query time, but it keeps the top-10 averages in a relatively narrow 0.54–0.56 band by giving similar URLs more chances to collide in at least one band.

## 4 Brute-force Baseline (Task 2)

For the same 200 queries, I measure the cost of scanning the entire corpus: for each query $q$, I compute 3-gram Jaccard similarity with every other URL in the dataset. This yields the exact nearest neighbors and provides a point of comparison for LSH.

Let $n$ be the number of unique URLs. The brute-force driver reports:

- the total wall-clock time to process all 200 queries;
- the average time per query;

- an estimate of the time required to compute all $\binom{n}{2}$ pairwise Jaccard similarities, extrapolating from the measured per-pair cost.

| Metric | Value |
|---|---|
| Brute-force total time for 200 queries | 102.97 seconds |
| Brute-force time per query (seconds) | 0.5148 |
| Estimated time for all $\binom{n}{2}$ pairs (seconds) | 97,269.13 (approximately 27.0 hours) |

Table 4: Brute-force Jaccard baseline timing (Task 2).

Even though brute-force gives exact Jaccard scores, its cost grows linearly in $n$ per query and quadratically for all pairs. In my implementation, the average LSH lookup time ($\approx 0.59$ seconds per query) is on the same order as the brute-force time ($\approx 0.51$ seconds), but LSH achieves this while probing only a small subset of candidates instead of scanning all 377,870 URLs. As $n$ grows larger, this sublinear candidate set makes LSH much more scalable than a full-corpus scan.

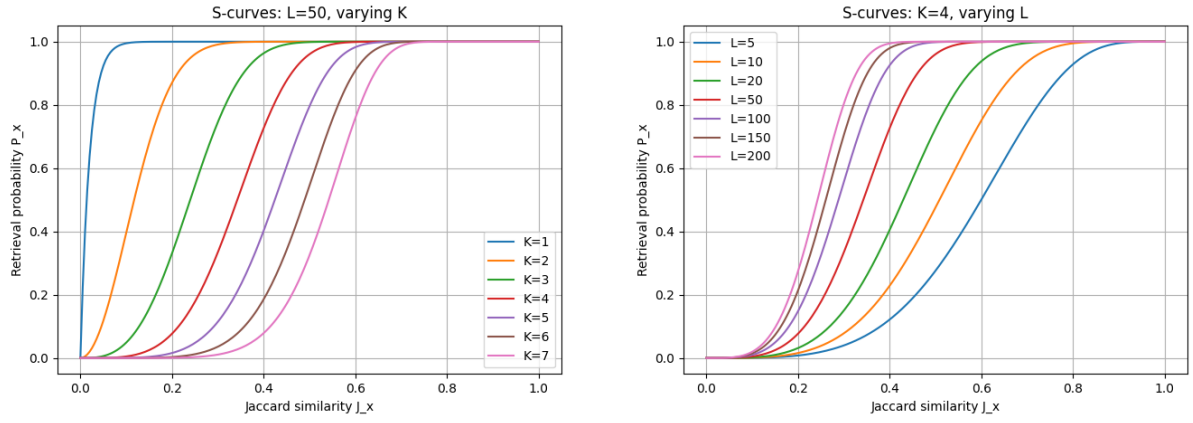
## 5 S-curve Analysis (Task 4)

For the theoretical MinHash LSH model, the probability of retrieving a pair with Jaccard similarity $J_x$ under parameters $(K, L)$ is

$$P_x = 1 - (1 - J_x^K)^L.$$

I sample $J_x$ on a dense grid from 0 to 1 and plot $P_x$ for two experimental sweeps:

- fixing $L = 50$ and varying $K \in \{1, 2, 3, 4, 5, 6, 7\}$;
- fixing $K = 4$ and varying $L \in \{5, 10, 20, 50, 100, 150, 200\}$.




(a) Fixed $L = 50$, varying $K$.

(b) Fixed $K = 4$, varying $L$.

Figure 1: Theoretical S-curves of the LSH retrieval probability for selected (K,L) configurations.

As $K$ increases (Figure 1a), the S-curve steepens and the "threshold" Jaccard at which $P_x$ rises sharply shifts to the right: higher $K$ rejects more low-similarity pairs but also requires URLs to be more similar before they are likely to collide in at least one band. Increasing $L$ while holding $K$ fixed (Figure 1b) raises the entire curve, boosting the probability of retrieving high-$J_x$ pairs without changing the threshold location much; this matches the intuition that more bands give more chances for a truly similar pair to agree in all $K$ coordinates of some band.

# 6 Adaptive Sampling vs Random Sampling (Task 2)

Let $S = \{w_1, \ldots, w_n\}$ be a multiset of reals and

$$T = \sum_{i=1}^{n} w_i$$

be the quantity we would like to estimate. We sample indices $i$ from a distribution $D$ over $\{1, \ldots, n\}$ with probabilities $p_i > 0$ and $\sum_i p_i = 1$, and define the importance-sampling estimator

$$\hat{T} = \frac{1}{k} \sum_{j=1}^{k} \frac{w_{I_j}}{p_{I_j}},$$

where $I_1, \ldots, I_k$ are i.i.d. draws from $D$.

Note that the assignment handout writes $\hat{T} = \sum_{j=1}^{k} w_{I_j}/p_{I_j}$ (without the $1/k$ factor). That expression has expectation $kT$ and is therefore biased; in what follows I analyze the standard unbiased importance-sampling estimator with the averaging factor $1/k$.

## 6.1 Unbiasedness of $\hat{T}$

Consider a single draw $I$ with $\Pr[I = i] = p_i$, and define

$$X = \frac{w_I}{p_I}.$$

Then

$$\mathbb{E}[X] = \sum_{i=1}^{n} \Pr[I = i] \cdot \frac{w_i}{p_i} = \sum_{i=1}^{n} p_i \cdot \frac{w_i}{p_i} = \sum_{i=1}^{n} w_i = T.$$

So the one-sample estimator $X$ is unbiased for $T$.

For $k$ i.i.d. samples, $\hat{T}$ is the average of $X_1, \ldots, X_k$, where each $X_j$ is distributed like $X$:

$$\mathbb{E}[\hat{T}] = \mathbb{E}\left[\frac{1}{k} \sum_{j=1}^{k} X_j\right] = \frac{1}{k} \sum_{j=1}^{k} \mathbb{E}[X_j] = \frac{1}{k} \cdot k \cdot T = T.$$

Thus $\hat{T}$ is an unbiased estimator of $T$.

## 6.2 Variance of $\hat{T}$

We first compute the variance of a single-sample estimator $X$ and then scale by $k$. Since $\mathbb{E}[X] = T$,

$$\mathbb{E}[X^2] = \sum_{i=1}^{n} p_i \left(\frac{w_i}{p_i}\right)^2 = \sum_{i=1}^{n} \frac{w_i^2}{p_i},$$

$$\mathrm{Var}(X) = \mathbb{E}[X^2] - (\mathbb{E}[X])^2 = \sum_{i=1}^{n} \frac{w_i^2}{p_i} - T^2.$$

The $X_j$ are independent and identically distributed, so

$$\mathrm{Var}(\hat{T}) = \mathrm{Var}\left(\frac{1}{k} \sum_{j=1}^{k} X_j\right) = \frac{1}{k^2} \sum_{j=1}^{k} \mathrm{Var}(X_j) = \frac{1}{k} \mathrm{Var}(X).$$

Therefore

$$\boxed{\mathrm{Var}(\hat{T}) = \frac{1}{k} \left(\sum_{i=1}^{n} \frac{w_i^2}{p_i} - T^2\right).}$$

### 6.3 Designing $p_i$ vs uniform sampling

Under uniform random sampling, $p_i = 1/n$ for all $i$, so

$$\sum_{i=1}^{n} \frac{w_i^2}{p_i} = \sum_{i=1}^{n} w_i^2 \cdot n = n \sum_{i=1}^{n} w_i^2,$$

and the variance becomes

$$\mathrm{Var}_{\mathrm{uniform}}(\hat{T}) = \frac{1}{k} \left( n \sum_{i=1}^{n} w_i^2 - T^2 \right).$$

To minimize the variance over all valid probability vectors $p_i > 0$ with $\sum_i p_i = 1$, we minimize $\sum_i w_i^2/p_i$ subject to this constraint. Using Lagrange multipliers,

$$\mathcal{L}(p, \lambda) = \sum_{i=1}^{n} \frac{w_i^2}{p_i} + \lambda \left( \sum_{i=1}^{n} p_i - 1 \right).$$

Setting $\partial \mathcal{L}/\partial p_i = 0$ gives

$$-\frac{w_i^2}{p_i^2} + \lambda = 0 \quad \Rightarrow \quad p_i^2 = \frac{w_i^2}{\lambda} \quad \Rightarrow \quad p_i \propto |w_i|.$$

When all $w_i \geq 0$, the optimal distribution is

$$p_i^\star = \frac{w_i}{\sum_{j=1}^{n} w_j} = \frac{w_i}{T}.$$

Plugging this into the variance term,

$$\sum_{i=1}^{n} \frac{w_i^2}{p_i^\star} = \sum_{i=1}^{n} \frac{w_i^2}{w_i/T} = \sum_{i=1}^{n} w_i \cdot T = T^2,$$

so

$$\mathrm{Var}_{\mathrm{optimal}}(\hat{T}) = \frac{1}{k}(T^2 - T^2) = 0.$$

In other words, if we somehow knew the exact $\{w_i\}$ and $T$ ahead of time and sampled with $p_i \propto w_i$, the importance-sampling estimator would have zero variance and always output $T$ exactly. In practice we do not know $T$, but this analysis shows that we can reduce variance below that of uniform sampling by allocating higher sampling probability to larger weights $w_i$. Unless all $w_i$ are equal, there always exists a non-uniform choice of $p_i$ that strictly improves on the uniform-variance baseline.

# A How to Run the Code

- Files in the working directory:
  - `MinHash.py`
  - `user-ct-test-collection-01.txt` (AOL query log)

### Commands

```
pip install numpy pandas matplotlib
python3 MinHash.py
```

Running `python3 MinHash.py` once produces:

- All timing and Jaccard values reported in Tables 1, 2, 3, and 4.
- The S-curve plots `s_curve_K_varies.png` and `s_curve_L_varies.png`.

# B Random Seeds and Deterministic Settings

### Key fixed parameters

- Precomputed MinHash signature length: 600 coordinates per URL.
- Baseline LSH (Tasks 1–2): $K = 2$, $L = 50$, $B = 64$, bucket range $R = 2^{20}$.
- Grid search (Task 3): $K \in \{2, 3, 4, 5, 6\}$, $L \in \{20, 50, 100\}$, with $B = 64$ and the same bucket range $R = 2^{20}$.
- Number of queries: 200.
- Unique URLs (after deduplication): 377,870.

### Seeds and determinism

- Query sampling:
  - RNG: `np.random.default_rng(42)` to select 200 query indices without replacement.
- MinHash signatures:
  - Deterministic `fast_minhash_code(i, shingle)` (no RNG).
- LSH bucket hashing:
  - Deterministic SHA-256–based `stable_hash((table_id, band_vals))` reduced modulo $R$ and then folded modulo $B$ to obtain the bucket index.

All results in the report are from a single run with these settings and are reproducible under the same code and dataset.