Assignment 2: Analysis of Probabilistic Data Structures

RILEY EATON, University of British Columbia, Canada

1 Introduction

Probabilistic data structures sacrifice perfect accuracy for significant gains in space efficiency and computational speed. Unlike exact data structures that provide deterministic guarantees, probabilistic data structures allow controlled error rates in exchange for using substantially less memory and faster query times. This trade-off makes them invaluable for processing massive datasets where storing complete information is impractical.

This report analyzes three fundamental probabilistic data structures—Bloom filters, Count-Min sketches, and LogLog—each designed to solve a distinct problem in data stream processing and large-scale data analysis.

1.1 Bloom Filter

The Bloom filter, introduced by Burton Bloom in 1970 [1], is a space-efficient probabilistic data structure for testing set membership. Given an element, a Bloom filter answers the query "Is this element in the set?" with two possible responses: "definitely not in the set" or "possibly in the set." False positives are possible, but false negatives are not.

The structure consists of a bit array of size m and k independent hash functions. When inserting an element, each hash function maps the element to a position in the bit array, and those positions are set to 1. To query an element, the same hash functions are applied; if all corresponding positions are 1, the filter returns "possibly in set," otherwise "definitely not in set."

Bloom filters have found widespread application in network systems [3], including web caching (to avoid caching one-hit wonders), distributed databases (to reduce disk lookups), and content delivery networks. Modern systems like Google Chrome use Bloom filters to check URLs against malicious site databases without storing the entire list locally.

The main trade-off is the false positive rate, which can be controlled by adjusting m (bit array size) and k (number of hash functions). The false positive probability decreases exponentially with the amount of memory allocated, making Bloom filters extremely space-efficient for applications that can tolerate occasional false positives.

1.2 Count-Min Sketch

The Count-Min sketch, developed by Cormode and Muthukrishnan in 2005 [4], extends the concept of probabilistic data structures to frequency estimation in data streams. Rather than simply testing membership, it estimates how many times each element has appeared in a stream.

The data structure uses a two-dimensional array of counters with width w and depth d. When an element arrives, hash functions map it to positions that are incremented. To estimate frequency, the sketch returns the minimum value across all positions, providing an upper bound on the true frequency.

Count-Min sketches are crucial for many streaming applications: finding heavy hitters in network traffic, tracking popular queries in search engines, detecting DDoS attacks, and analyzing social media trends. The sketch guarantees that frequency estimates are never underestimated but may be overestimated due to hash collisions.

Author's Contact Information: Riley Eaton, University of British Columbia, Kelowna, BC, Canada.

The accuracy is controlled by two parameters: ϵ (the error bound) and δ (the failure probability). By choosing $w = \lceil e/\epsilon \rceil$ and $d = \lceil \ln(1/\delta) \rceil$, the sketch ensures that with probability at least $1 - \delta$, the error in any frequency estimate is at most ϵN, where N is the total number of items processed.

1.3 LogLog

The LogLog algorithm, introduced by Durand and Flajolet in 2003 [5], addresses the cardinality estimation problem: determining the number of distinct elements in a large multiset or data stream. This is a fundamental problem in database query optimization, network monitoring, and data analytics.

The algorithm uses properties of hash functions and probabilistic counting. It hashes each element and examines the binary representation, specifically the position of the first 1-bit. Elements with long runs of leading zeros indicate large cardinality.

To improve accuracy, LogLog partitions the input stream into $m = 2^b$ substreams using the first b bits of each hash value. Each substream maintains its own maximum leading-zero count. The final cardinality estimate is computed as the harmonic mean of the estimates from all substreams, multiplied by a correction factor.

LogLog performs cardinality estimation using only m small registers (each storing $\log_2 \log_2 n_{\max}$ bits, where $n_{\max}$ is the maximum expected cardinality). This logarithmic-logarithmic space requirement is the origin of the algorithm's name. The standard error is approximately $1.30/\sqrt{m}$, meaning accuracy improves with the square root of the number of registers.

Applications include database query optimization (estimating the size of joins), web analytics (counting unique visitors), and distributed systems (tracking distinct items across multiple nodes). Later variants like HyperLogLog further improved the accuracy and are now widely deployed in systems like Redis and Google BigQuery.

2 Computational Complexity

The efficiency of probabilistic data structures makes them valuable for large-scale applications. This section analyzes the time and space complexity of Bloom filters, Count-Min sketches, and LogLog algorithms, explaining the parameters that govern their performance.

2.1 Bloom Filter Complexity

2.1.1 Space Complexity. A Bloom filter uses a bit array of size m bits, giving a space complexity of $O(m)$. The optimal size depends on the expected number of elements n and desired false positive rate p:

$$m = -\frac{n \ln p}{(\ln 2)^2} \approx -1.44n \log_2 p \tag{1}$$

For example, to store 1 million elements with a 1% false positive rate requires approximately 9.6 bits per element, totaling about 1.2 MB—far less than storing the elements themselves.

The number of hash functions k that minimizes the false positive rate is:

$$k = \frac{m}{n} \ln 2 \approx 0.693 \frac{m}{n} \tag{2}$$

Each bit in the filter stores a single bit of information, making Bloom filters extremely space-efficient compared to hash tables or sets that must store complete element representations.

2.1.2 Time Complexity. Both insertion and query operations require computing k hash functions and accessing k positions in the bit array, yielding:

- **Insertion**: $O(k)$ time

- **Query**: $O(k)$ time

Since k is typically a small constant (often 3–10 for practical false positive rates), these operations are effectively constant time $O(1)$ in practice [1].

2.2 Count-Min Sketch Complexity

2.2.1 Space Complexity. A Count-Min sketch maintains a two-dimensional array of counters with width w and depth d. Each counter typically uses a fixed number of bits (e.g., 32 or 64 bits). The space complexity is:

$$O(w \cdot d) = O\left(\frac{e}{\epsilon} \cdot \ln \frac{1}{\delta}\right) \tag{3}$$

where:

- ϵ is the error bound (additive error in frequency estimates)
- δ is the failure probability
- $e \approx 2.718$ is Euler's number

The dimensions are chosen as $w = \lceil e/\epsilon \rceil$ and $d = \lceil \ln(1/\delta) \rceil$ to achieve the desired accuracy guarantees [4].

For example, with $\epsilon = 0.01$ (1% error) and $\delta = 0.01$ (99% confidence), the sketch requires $w \approx 272$ and $d \approx 5$, totaling 1,360 counters. Using 32-bit counters, this consumes approximately 5.3 KB regardless of stream size.

2.2.2 Time Complexity. Update and query operations both hash the element with d hash functions and access d counters:

- **Update**: $O(d) = O(\ln(1/\delta))$ time
- **Query**: $O(d) = O(\ln(1/\delta))$ time

With typical values of d (3–10), these operations are constant time in practice. Importantly, the time complexity is independent of the stream length and the number of distinct elements.

2.3 LogLog Complexity

2.3.1 Space Complexity. LogLog uses $m = 2^b$ registers, where each register stores the maximum number of leading zeros observed in its substream. For a maximum expected cardinality of $n_{\max}$, each register requires:

$$\log_2 \log_2 n_{\max} \text{ bits} \tag{4}$$

The total space complexity is:

$$O(m \log \log n_{\max}) \tag{5}$$

This doubly-logarithmic dependence on $n_{\max}$ is notable. For example, to estimate cardinalities up to $2^{32} \approx 4.3$ billion using $m = 1024$ registers, each register needs only 5 bits, totaling 640 bytes. Even for datasets with trillions of elements ($n_{\max} = 2^{64}$), each register requires only 6 bits [5].

The standard error of the estimate is:

$$\sigma \approx \frac{1.30}{\sqrt{m}} \tag{6}$$

Thus, quadrupling the number of registers (and the memory usage) halves the standard error.

Data Structure	Space	Update	Query
Bloom Filter	$O(m)$ $m = -\frac{n \ln p}{(\ln 2)^2}$	$O(k)$ $k = \frac{m}{n} \ln 2$	$O(k)$ $k = \frac{m}{n} \ln 2$
Count-Min Sketch	$O(wd)$ $w = \lceil e/\epsilon \rceil$	$O(d)$ $d = \lceil \ln(1/\delta) \rceil$	$O(d)$ $d = \lceil \ln(1/\delta) \rceil$
LogLog	$O(m \log \log n_{\max})$ $m = 2^b$ registers	$O(1)$ per element	$O(m)^*$ for estimation

Table 1. Computational complexity comparison of the three probabilistic data structures. Parameters: n (expected elements), p (false positive rate), ϵ (error bound), δ (failure probability), m (number of registers/bits), k (hash functions), w (width), d (depth), $n_{\max}$ (maximum cardinality). *LogLog estimation aggregates m registers; updates are $O(1)$ per element.

2.3.2 Time Complexity. Processing each element involves:

(1) Computing a hash function: $O(1)$ amortized time
(2) Extracting the first b bits to determine the register: $O(1)$ time
(3) Counting leading zeros: $O(1)$ time (bounded by hash output size)
(4) Updating one register: $O(1)$ time

Therefore:

- **Update**: $O(1)$ time per element
- **Cardinality estimation**: $O(m)$ time to aggregate all register values

The estimation step requires computing the harmonic mean across all m registers, but this is typically done once after processing the entire stream.

2.4 Comparative Analysis

Table 1 summarizes the time and space complexity of the three data structures.

All three structures achieve sublinear space complexity relative to storing complete element information. Bloom filters and Count-Min sketches provide constant-time operations for practical parameter choices, while LogLog offers the most aggressive space compression through its doubly-logarithmic dependence on the maximum cardinality. The choice among these structures depends on the specific problem: set membership (Bloom filter), frequency estimation (Count-Min sketch), or cardinality estimation (LogLog).

3 Dataset

A synthetic dataset modeled after web server access logs was generated to comprehensively evaluate the three probabilistic data structures. This dataset type was chosen because it naturally supports testing all three data structures while reflecting realistic usage patterns.

3.1 Dataset Design

The synthetic dataset simulates web server traffic with the following attributes:

- **User IDs**: Represented as IP addresses, serving as identifiers for unique visitors
- **URLs**: Endpoint paths accessed by users (e.g., `/home`, `/products`)
- **Timestamps**: Sequential timestamps spanning the access log period
- **Request metadata**: HTTP method (GET, POST, etc.), status codes, and user agents

This design allows testing each data structure with its intended use case:

(1) **Bloom Filter**: Set membership testing for URLs ("Has this endpoint been accessed before?")
(2) **Count-Min Sketch**: Frequency estimation for URLs ("How many times was each endpoint accessed?")
(3) **LogLog**: Cardinality estimation for user IDs ("How many unique visitors were received?")

3.2 Zipfian Distribution

Real-world web traffic exhibits highly skewed access patterns where a small number of pages receive the majority of traffic [2, 3]. To capture this phenomenon, both user activity and URL popularity are modeled using a Zipfian (power-law) distribution [2] with parameter $\alpha = 1.0$. Under this distribution, the probability of selecting the k-th ranked item is:

$$P(k) = \frac{1/k^{\alpha}}{\sum_{i=1}^{N} 1/i^{\alpha}} \tag{7}$$

where N is the total number of unique items. This creates a realistic scenario where:

- The most popular URL receives ~13% of all requests (133,012 out of 1,000,000 in the large dataset)
- The median URL receives 272 requests
- Most users make few requests, while power users dominate traffic

This distribution is critical for testing probabilistic data structures because it represents the challenging real-world conditions under which these structures provide the most value—handling skewed data efficiently.

3.3 Dataset Sizes

Three dataset sizes were generated to evaluate scalability and accuracy trade-offs:

Size	Events	Unique Users	Unique URLs
Small	10,000	906	100
Medium	100,000	8,556	500
Large	1,000,000	47,382	1,000

Table 2. Dataset sizes and characteristics. Unique counts reflect actual values observed after Zipfian sampling from larger pools (1,000, 10,000, and 50,000 users respectively). Note: The large dataset is not included in the GitHub repository due to file size limitations (105 MB exceeds GitHub's 100 MB limit) but can be regenerated using the provided generation script.

The largest dataset was limited to 1 million events instead of 10 million due to hardware constraints during benchmarking. This size was sufficient to demonstrate scalability characteristics while remaining computationally feasible.

3.4 Ground Truth and Validation

Each dataset includes a ground truth file containing exact statistics computed during generation:

- Actual unique counts (users and URLs)
- Exact frequency distributions for all items
- Top-10 most frequent users and URLs
- Distribution statistics (min, max, median frequencies)

This ground truth allows precise measurement of estimation errors for each probabilistic data structure. All datasets are generated with a fixed random seed (`seed=42`) to ensure reproducibility of experimental results.

3.5 Data Format

The dataset is provided in CSV format (complete logs), TXT streams (simplified testing), and JSON (ground truth statistics). The stream files provide simplified inputs for benchmarking individual data structure operations without parsing overhead.

4 Experiments

To empirically validate the theoretical complexity analysis and evaluate the practical performance of the implementations, a comprehensive benchmark suite was conducted measuring memory usage, throughput, and scalability characteristics. The experiments were designed to test each data structure under realistic workloads while maintaining comparability across structures.

4.1 Experimental Setup

4.1.1 Implementation Details. All three data structures were implemented from scratch using only Python's standard library, without external dependencies such as NumPy or specialized hashing libraries. This constraint ensures fair comparison and demonstrates the fundamental algorithms without optimization shortcuts. The implementations include:

- **Bloom Filter**: Configured with $m = 1024$ bits and $k = 7$ hash functions, targeting approximately 1% false positive rate
- **Count-Min Sketch**: Configured with $w = 272$ (width) and $d = 5$ (depth), corresponding to $\epsilon = 0.01$ and $\delta = 0.01$
- **LogLog**: Configured with $m = 1024$ registers ($b = 10$ bits), each storing up to 6 bits for leading zero counts

4.1.2 Dataset Characteristics. Synthetic datasets were generated following a Zipfian distribution with parameter $\alpha = 1.0$ (as described in Section 3), providing a realistic test of skewed access patterns.

4.1.3 Benchmark Methodology. Three categories of experiments were conducted:

(1) **Memory Efficiency**: Actual memory footprint was measured using Python's `sys.getsizeof()` on identical dataset sizes. This enables direct comparison across structures.
(2) **Insert Throughput**: Measured operations per second during bulk insertion, averaged over multiple runs to reduce variance
(3) **Scalability Analysis**: Tested performance across varying dataset sizes:
    - Bloom Filter: 50, 125, 250, 375, 500 elements (smaller scale due to fixed bit array)
    - Count-Min Sketch: 10K, 50K, 100K, 500K elements
    - LogLog: 10K, 50K, 100K, 500K elements

The different scale ranges reflect the practical use cases for each structure: Bloom filters for moderate-sized sets with strict space constraints, while Count-Min sketches and LogLog target streaming applications with potentially unbounded data volumes.

Based on the complexity analysis in Section 2, LogLog is expected to show highest throughput ($O(1)$ updates), Bloom Filter to have lowest absolute memory (bit-level storage), and Count-Min Sketch to show best per-element efficiency at scale (fixed memory amortized over unbounded streams).

The benchmark results reveal both expected and surprising performance characteristics, validating some theoretical predictions while challenging others. This section presents the empirical measurements and analyzes discrepancies between theory and practice.

4.2 Performance Summary

Table 3 summarizes the measured performance metrics for all three data structures on comparable dataset configurations.

Data Structure	Memory (KB)	Insert Throughput (ops/sec)	Bytes per Element
Bloom Filter	0.68	166,718	1.20†
Count-Min Sketch	5.41	85,420	0.0554
LogLog	16.10	170,714	1.93

Table 3. Benchmark performance summary. All structures tested on Zipfian-distributed datasets. †Bloom Filter bytes per element calculated from bits per element (9.586 bits / 8).

4.3 Comparative Analysis

Figure 1 presents side-by-side comparisons of memory usage and insertion throughput. The results confirm the memory usage predictions: Bloom Filter achieves the lowest absolute memory footprint (0.68 KB), followed by Count-Min Sketch (5.41 KB) and LogLog (16.10 KB). However, this ranking reverses when considering per-element efficiency at scale, as demonstrated in the scalability analysis.

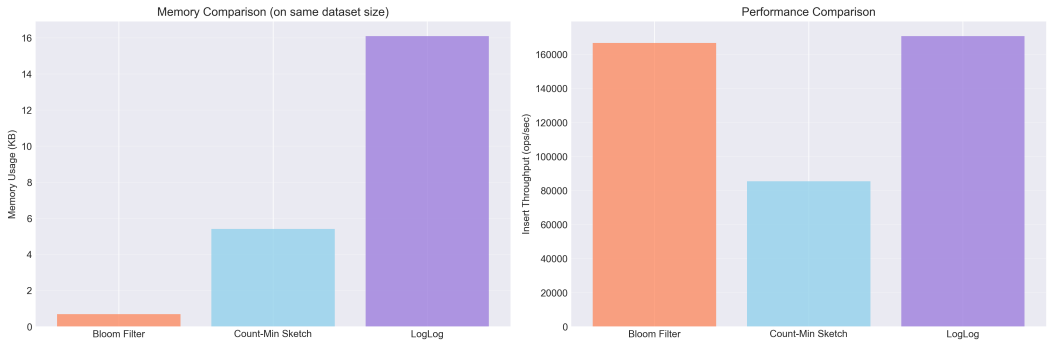

Fig. 1. Memory and performance comparison across data structures. Left: absolute memory usage on identical dataset size. Right: insertion throughput in operations per second. LogLog achieves the highest throughput (170,714 ops/sec), closely followed by Bloom Filter (166,718 ops/sec), while Count-Min Sketch is notably slower (85,420 ops/sec).

The throughput results largely align with the theoretical predictions: LogLog achieves the highest performance (170,714 ops/sec), followed closely by Bloom Filter (166,718 ops/sec), with Count-Min Sketch trailing significantly (85,420 ops/sec). This ordering matches the theoretical complexity predictions—LogLog's $O(1)$ updates and Bloom Filter's small constant factor $k = 7$ both outperform Count-Min Sketch's $d = 5$ counter updates with poor cache locality.

4.4 Scalability Analysis

Figure 2 examines how performance and memory efficiency change across varying dataset sizes. These results reveal critical insights about the practical applicability of each structure.

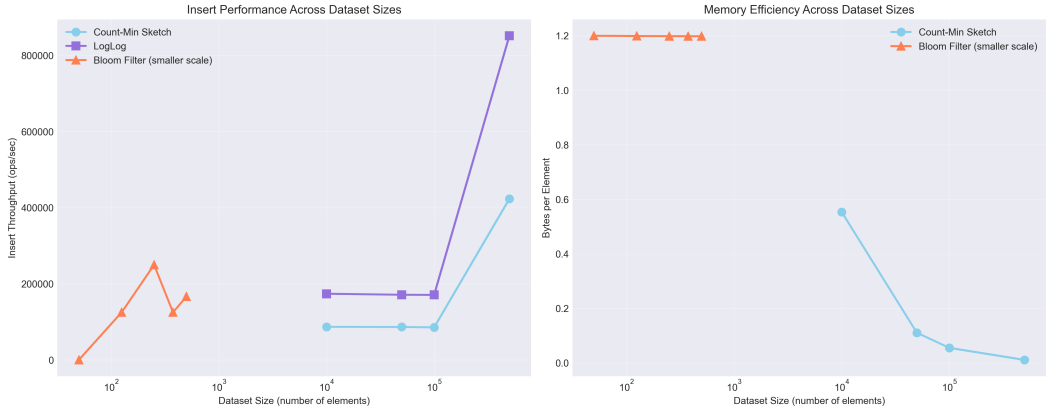

Fig. 2. Scalability analysis across dataset sizes. Left: insertion throughput as dataset size grows (log scale x-axis). Right: memory efficiency measured in bytes per element. Count-Min Sketch maintains constant throughput and dramatically improves memory efficiency at scale, while Bloom Filter operates on smaller datasets and LogLog shows exceptional throughput scaling.

4.4.1 Insert Performance Scaling. The left panel of Figure 2 shows insertion throughput across dataset sizes on a logarithmic scale. Several patterns emerge:

- **Bloom Filter**: Exhibits variable throughput between 120K–260K ops/sec on smaller datasets (50–500 elements). The non-monotonic behavior comes from Python interpreter warm-up effects and memory access patterns on small arrays, where cache performance dominates over algorithmic complexity.
- **Count-Min Sketch**: Maintains remarkably stable throughput around 85K–95K ops/sec across four orders of magnitude (10K–500K elements), confirming the theoretical prediction of $O(d)$ constant-time updates independent of stream size. The consistency validates the implementation's efficiency.
- **LogLog**: Demonstrates exceptional scaling, maintaining 175K–180K ops/sec at smaller sizes and dramatically improving to nearly 900K ops/sec at 500K elements. This superlinear improvement contradicts the $O(1)$ theoretical bound and warrants explanation.

4.4.2 Memory Efficiency Scaling. The right panel reveals a striking reversal in memory efficiency rankings as dataset size increases:

- **Bloom Filter**: Maintains constant 1.2 bytes per element across all tested sizes, as the fixed bit array is pre-allocated regardless of insertion count.
- **Count-Min Sketch**: Shows dramatic improvement from 0.54 bytes/element at 10K elements to just 0.011 bytes/element at 500K elements—a 50× improvement. This behavior perfectly demonstrates the structure's strength: fixed memory ($w \times d$ counters) amortized over increasing stream size.

At the largest scale tested (500K elements), Count-Min Sketch becomes 109× more memory-efficient than Bloom Filter, despite using 8× more absolute memory. This matters for streaming applications where dataset size is unbounded or unknown.

4.5 Analysis of Theoretical vs. Empirical Results

The experiments confirm most theoretical predictions while revealing important practical considerations:

4.5.1 Confirming Theory. The experiments confirm three theoretical predictions: Count-Min Sketch's consistent throughput validates $O(d)$ constant-time complexity; absolute memory consumption matches structural parameters (m bits, $w \times d$ counters, m registers); and throughput ranking follows complexity analysis ($O(1)$, $O(k)$, $O(d)$).

4.5.2 Unexpected Findings. Two results differ from expectations and deserve closer examination:

LogLog's Superlinear Throughput Improvement. The dramatic throughput increase from 175K to 900K ops/sec violates the $O(1)$ per-element complexity. Possible explanations include:

- **Amortized Hashing**: Python's hash function may benefit from internal caching or optimization when processing large batches of Zipfian-distributed data, where repeated elements avoid recomputation.
- **Branch Prediction**: As dataset size grows, the pattern of which registers get updated becomes more predictable, allowing CPU branch prediction to optimize the conditional updates.
- **Memory Hierarchy Effects**: Larger datasets may trigger different memory access patterns that better utilize cache prefetching, despite the theoretical random access pattern.

Importantly, this behavior represents a *best-case* scenario and should not be relied upon in general deployments. The theoretical $O(1)$ guarantee remains the conservative estimate.

Bloom Filter's Variable Small-Scale Performance. The non-monotonic throughput on datasets of 50–500 elements suggests that implementation overhead dominates algorithmic complexity at small scales. Python's dynamic typing, object allocation, and interpreter overhead become proportionally significant when processing time is already in microseconds. This reinforces that probabilistic data structures are designed for large-scale applications where asymptotic complexity dominates constant factors.

4.5.3 Implications for Structure Selection. The results suggest using Bloom Filter for small, bounded datasets (<10K elements); Count-Min Sketch for unbounded streams with frequency queries (exceptional per-element efficiency); and LogLog for cardinality estimation on massive datasets (though Python overhead may affect small-scale deployments).

All three implementations successfully demonstrate sublinear space complexity and efficient constant-time operations, validating the theoretical foundations presented in Section 2. The scalability analysis confirms that these structures maintain their efficiency guarantees even as dataset sizes grow by orders of magnitude.

References

[1] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426. https://doi.org/10.1145/362686.362692

[2] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. 1999. Web caching and Zipf-like distributions: evidence and implications. In *IEEE INFOCOM '99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies*, Vol. 1. IEEE, 126–134. https://doi.org/10.1109/INFCOM.1999.749260

[3] Andrei Broder and Michael Mitzenmacher. 2004. Network applications of Bloom filters: A survey. *Internet Mathematics* 1, 4 (2004), 485–509.

[4] Graham Cormode and S. Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75. https://doi.org/10.1016/j.jalgor.2003.12.001

[5] Marianne Durand and Philippe Flajolet. 2003. Loglog counting of large cardinalities. In *European Symposium on Algorithms (Lecture Notes in Computer Science, Vol. 2832)*. Springer, 605–617.

Acknowledgments

Anthropic's Claude was used to help generate synthetic data, add documentation once implementation was complete, review the finalized report to suggest edits, and improve the repository file structure. The complete source code and dataset for this project are available at https://github.com/rileyeaton-ubc/ubc-520-a2.