A Learning Augmented Approach to Cardinality Estimation

Mărcuș Alexandru Marian

January 14, 2026

Abstract

Cardinality estimation is the problem of determining the number of distinct elements in a data stream, useful for many tasks in database systems and network monitoring. This paper presents a deep learning approach that augments the classical probabilistic sketch with a lightweight Multilayer Perceptron. Rather than replacing the analytical model entirely, our neural network learns to predict the logarithmic residual error of the HyperLogLog estimate, acting as a continuous, learned bias-correction function. Experiments on synthetic data show that the model successfully unifies the different methods used for each cardinality range by this algorithm. Furthermore, on the real-world Shakespeare dataset, our approach achieves a relative error of $\approx 0.04\%$, outperforming the standard HLL algorithm while maintaining the memory and time complexity of the original data structure.

1 Introduction

Determining the number of distinct elements in a data stream is an active research area, useful in a wide range of fields in computer science. While the cardinality can be computed exactly using hash sets or sorting, for many applications, this is impractical due to linear memory and time requirements. For tasks where only an approximation of the cardinality is needed, there have been developed algorithms that require significantly fewer resources. Applications of such algorithms include database systems, that use them for query processing and optimization, network security monitoring and search engines [1]. Various algorithms have been proposed for this, including HyperLogLog [2], which has the advantage of being parallelizable while computing the cardinality estimate in a single pass of the input data. HLL is theoretically elegant, but its practical implementations are more complex. The standard asymptotic error formula ($\frac{1.04}{\sqrt{m}}$) fails for small cardinalities, necessitating a switch to a Linear Counting algorithm [2]. Furthermore, in the transition zones—where the cardinality is neither small enough for Linear Counting nor large enough for the asymptotic formula, HLL performs badly. Current state-of-the-art implementations attempt to mitigate this by storing large, pre-computed lookup tables and sparse arrays for bias correction [3]. These heuristics are complex to implement. Our solution aims to have a machine learning model learn the true mapping from sketch states to cardinality directly from data, bypassing the need for human-derived heuristics.

2 Related work

The problem of cardinality estimation has been extensively studied due to its critical importance in database query optimization and network traffic monitoring. Early probabilistic approaches, such as Linear Counting [4], relied on counting empty buckets in a hash table. While highly accurate for small cardinalities, Linear Counting requires memory linear to the cardinality, making it impractical for large datasets. To address high-cardinality streams, Flajolet et al. introduced the HyperLogLog algorithm [2], which achieves a standard error of $1.04/\sqrt{m}$ using only $O(\log \log N)$ memory. HLL relies on the observation that the maximum number of leading zeros in the binary representation of hashed elements serves as a good indicator of cardinality. However, the raw HLL estimator exhibits significant bias for small cardinalities and requires switching to Linear Counting when the number of empty registers is high.

Algorithmic Engineering. The practical limitations of the asymptotic HLL formula led to the development of HyperLogLog++ by Heule et al. [3]. This implementation introduces complex heuristics: a sparse representation for very small sets, and an empirical lookup tables to correct bias, especially in the transition zone between Linear Counting and the asymptotic regime. While effective, HLL++ increases implementation complexity and relies on fixed, pre-computed constants that may not be optimal for all distributions. More recently, Ertl [5] proposed a rigorous mathematical solution using Maximum Likelihood Estimation. This eliminates the need for lookup tables and achieves lower variance than HLL++, but it requires finding the roots of a complex function at query time, which can be computationally expensive.

Learned Algorithms. Following Kraska et al. paper [6] there has been a trend of replacing heuristic-based system components with machine learning models. This work showed that neural networks could learn the cumulative distribution function of data to outperform traditional B-Trees, inspiring further research into "learned" database components. Closest to our work is Wu et al., who proposed a supervised learning framework for cardinality estimation [7]. They demonstrated that a deep learning model could effectively estimate cardinality using a sample based method. Our work is different, focusing on enhancing the already existing sketch based method HLL with the help of machine learning.

In this paper we propose a correction model that acts as a lightweight replacement for the HLL++ need for lookup tables, linear counting and sparse representation of the registers, preserving the standard HLL sketch format while smoothing the discontinuity between Linear Counting and the asymptotic regime.

3 Preliminaries

3.1 The HyperLogLog Algorithm

The HyperLogLog algorithm approximates the cardinality of a multiset by analyzing the statistical properties of hashed elements. It relies on the observation that in a stream of uniformly distributed random integers, the maximum number of leading zeros in the binary representation is correlated with the number of distinct elements in the stream. To reduce the variance of a single observation, it uses a technique known as stochastic averaging [8]. The input stream S is partitioned into $m = 2^p$ substreams S_i using the first p bits of the 32-bit hash value. The remaining bits are used to compute the number of leading zeros in the binary representation of the hashed value. The algorithm maintains a state vector (or "sketch") of m registers, $M = [M_1, \ldots, M_m]$, where each register M_j stores the maximum rank observed in the i-th substream:

$$M_j = \max_{x \in S_i} \rho(x)$$

where $\rho(x)$ denotes the number of leading zeros in the binary representation of x plus one. By convention, $max_{x \in \emptyset} \rho(x) = -\infty$. The algorithm uses these registers to compute the cardinality estimate as the normalized bias corrected harmonic mean of the estimations on the substreams

$$E := \alpha_m m^2 \left(\sum_{j=1}^{m} 2^{-M_j} \right)^{-1}$$

Here, α_m is a bias-correction constant derived from the integral of the log-distribution.

$$\alpha_m := \left(m \int_0^\infty \left(\log_2 \left(\frac{2+u}{1+u} \right) \right)^m du \right)^{-1}$$

More details about the algorithm can be found in the original paper by Flajolet et al. [2].

Algorithm 1 HyperLogLog

```
Require: Let $h : \mathcal{D} \to \{0,1\}^{32}$ hash data from domain $\mathcal{D}$. Let $m = 2^p$ with $p \in [4..16]$.
    Phase 0: Initialization.
1:  Define $\alpha_{16} = 0.673, \alpha_{32} = 0.697, \alpha_{64} = 0.709, \alpha_m = 0.7213/(1 + 1.079/m)$ for $m \geq 128$
2:  Initialize $m$ registers $M[0]$ to $M[m-1]$ to 0.
    Phase 1: Aggregation.
3:  for all $v \in S$ do
4:      $x := h(v)$
5:      $id := (x_{31}, \ldots, x_{32-p})_2$                    ▷ First $p$ bits of $x$
6:      $w := (x_{31-p}, \ldots, x_0)_2$
7:      $M[idx] := \max(M[idx], \rho(w))$
8:  end for
    Phase 2: Result computation.
9:  $E := \alpha_m m^2 \left( \sum_{j=1}^{m} 2^{-M_j} \right)^{-1}$
10: if $E \leq \frac{5}{2} m$ then
11:     Let $V$ be the number of registers equal to 0.
12:     if $V \neq 0$ then
13:         $E^* := m \log(m/V)$
14:     else
15:         $E^* := E$
16:     end if
17: else if $E \leq \frac{1}{30} 2^{32}$ then
18:     $E^* := E$
19: else
20:     $E^* := -2^{32} \log(1 - E/2^{32})$
21: end if
22: return $E^*$
```

3.2 Bias and Correction

While the estimator E is asymptotically unbiased, simulations by Flajolet et. al. show that for $n < \frac{5}{2}m$ nonlinear distortions appear. To address this, standard implementations like HyperLogLog++ [3] employ a hybrid approach that switches between different algorithms based on the estimated cardinality:

Linear Counting. When empty registers exist, HLL++ calculates the estimate for this range using the formula $E_LC = m\log(\frac{m}{V})$ [4], where V is the number of empty registers. Also, it uses a sparse format for in memory representation of the sketch when calculating small cardinalities.

Bias Correction. In the transition range where Linear Counting becomes inaccurate but E is still biased, HLL++ relies on empirical bias-correction lookup tables derived from simulation.

Large Range. As n approaches 2^{32}, collisions in the 32-bit hash space cause underestimation. HLL++ uses a 64-bit hash to solve this issue.

4 Methodology

4.1 Learned HLL

Instead of relying on heuristic lookup tables or discrete mode-switching, we propose a learning-based approach that treats bias correction as a regression problem.

Problem Formulation. Direct prediction of cardinality N from sketch registers is difficult due to the large output range (0 to 10^9). To stabilize learning, we formulate the problem as predicting the residual log-error of the raw estimator. Given a normalized histogram of register values H, our model f_θ predicts a correction factor c:

$$\log_{10}(c) = f_\theta(H)$$

The final estimated cardinality $\hat{N}$ is obtained by scaling the raw estimate:

$$\hat{N} = E \cdot 10^{f_\theta(H)}$$

This approach allows the model to focus purely on learning the bias pattern rather then the entire math behind the model. **Model Architecture.** We employ a lightweight Multi-Layer Perceptron (MLP) to ensure low inference latency. The input is a feature vector derived from the HLL registers. To capture the signal in the high-sparsity regime (where Linear Counting dominates), we augment the standard histogram with an occupancy feature, defined as $\log(1 - V/m)$, representing the density of non-empty registers. The network consists of three fully connected layers with Batch Normalization [9] and LeakyReLU [10] activations, minimizing a Huber Loss (SmoothL1) function. This architecture is differentiable and constant-time $O(1)$ relative to stream size, matching the efficiency requirements of streaming systems.

Algorithm 2 Learned HyperLogLog

```
Require: Hash function $h : \mathcal{D} \to \{0,1\}^{64}$
Require: Precision parameter $p \in [4..16]$, $m = 2^p$
Require: Pre-trained Neural Network $f_\theta$ (MLP)
    Phase 0: Initialization
 1: Initialize registers $M[0 \dots m-1] \leftarrow 0$
    Phase 1: Aggregation
 2: for all $v \in$ Stream $S$ do
 3:     $x \leftarrow h(v)$
 4:     $j \leftarrow$ integer$(x_{63} \dots x_{64-p})$          ▷ Register Index
 5:     $w \leftarrow (x_{63-p} \dots x_0)$                        ▷ Remaining Bits
 6:     $M[j] \leftarrow \max(M[j], \rho(w))$                       ▷ Update Rank
 7: end for
    Phase 2: Result Computation
 8: $Z \leftarrow \left(\sum_{j=0}^{m-1} 2^{-M[j]}\right)^{-1}$      ▷ Harmonic Sum
 9: $E \leftarrow \alpha_m \cdot m^2 \cdot Z$                       ▷ Standard HLL Estimate
10: $H \leftarrow$ Histogram$(M)$                                  ▷ Extract feature vector
11: $\delta \leftarrow f_\theta(H)$                                ▷ Predict log-residual correction
12: return $E \cdot 10^\delta$                                     ▷ Apply correction
```

4.2 Dataset Generation

To evaluate our approach, we generated a synthetic dataset containing $600,000$ distinct HyperLogLog sketches configured with a precision of $p = 16$. The true cardinality N for each sketch was sampled from a log-uniform distribution over the range $[10, 10^8]$, ensuring balanced representation across orders of magnitude.

To simulate the hash values of distinct elements, we generated uniformly distributed 64-bit integers using the **Xorshift64** algorithm [11]. This method allows us to simulate a high-quality hash function applied to unique stream elements, without the computational overhead of hashing string objects. The resulting dataset was partitioned into training (80%), validation (10%), and testing (10%) sets.

4.3 Training Strategy

Loss Function. Standard regression loss functions like Mean Squared Error are sensitive to outliers, which are common when estimating cardinality across orders of magnitude. To ensure robust convergence, we minimize the **Huber Loss** [12] (Smooth L1 Loss) between the predicted log-correction δ and the true log-residual $\delta_{true} = \log_{10}(N/E)$:

$$\mathcal{L}(\delta, \delta_{true}) = \begin{cases} \frac{1}{2}(\delta - \delta_{true})^2, & \text{if } |\delta - \delta_{true}| < 1 \\ |\delta - \delta_{true}| - \frac{1}{2}, & \text{otherwise} \end{cases}$$

This objective function behaves quadratically for small errors (providing smooth gradients) and linearly for large errors (preventing exploding gradients during the initial training phase).

Optimization. We train the network using the **AdamW** optimizer [13] with an initial learning rate of $5 \cdot 10^{-4}$ and a weight decay of 10^{-4} to prevent overfitting. We employ a dynamic learning rate scheduler that reduces the learning rate by a factor of 0.5 if the validation loss plateaus for 3 consecutive epochs.

Training Protocol. The model is trained for a maximum of 100 epochs with a batch size of 1024. To prevent overfitting to the synthetic data, we implement **Early Stopping** with a patience of 15 epochs, saving the model weights that achieve the lowest validation loss. All experiments were conducted on a single NVIDIA GPU using the PyTorch framework [14].

5 Experimental Results

5.1 Experimental Setup

To assess the quality of the cardinality estimates, we utilize the standard Relative Error metric, defined as:

$$\text{Error} = \frac{|\hat{N} - N|}{N}$$

where N is the true cardinality and $\hat{N}$ is the estimated cardinality. We analyze the error distribution across the entire logarithmic range of cardinalities to identify regime-specific behaviors.

For the HLL++ benchmarks, we utilized the $python - hll$ library [15], a Python port of the standard Java HLL implementation. This library includes the sparse representation and empirical bias correction logic defined in the HyperLogLog++ paper.

5.2 Accuracy Comparison

We compare our Learned HLL against two baselines:

1. Standard HLL (Theoretical): The raw harmonic mean estimator combined with the analytical Linear Counting lower bound.

2. HLL++: The production-grade implementation from the Google Guava library, which utilizes sparse representations and empirical bias correction.

Results. Figure 1 illustrates the relative error trends for all three methods. As shown in the figure, our learned model successfully captures the complex, non-linear bias inherent in the low-cardinality regime ($N < 5m$).

- Low Cardinality: The learned model (Red) closely follows the theoretical lower bound of Linear Counting (Green). While there is a small constant offset due to the resolution limits of the floating-point input features, the model correctly identifies the structural relationship between register occupancy and cardinality without explicit programming.

- Transition Zone: Unlike HLL++ (Blue), which exhibits a sharp discontinuity or "jump" when switching between Linear Counting and the raw estimator, the residual neural network provides a smooth, continuous transition. This eliminates the artifacts typically caused by heuristic mode-switching.

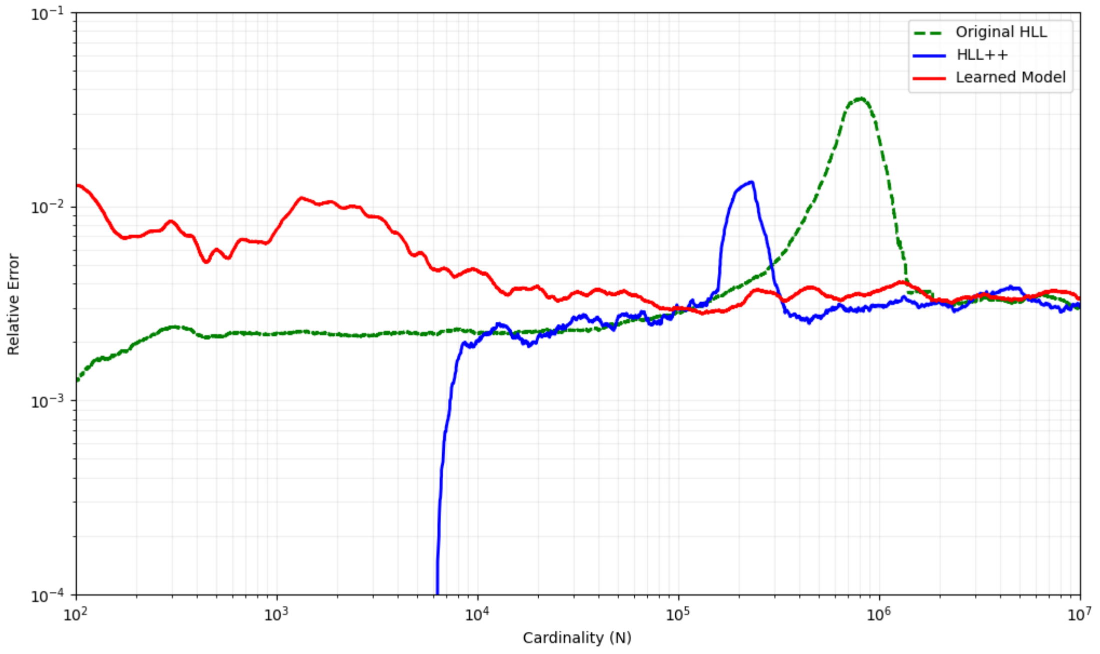

Figure 1: Comparison of Relative Error: Our approach (Red), HLL (Green), HLL++ (Blue). Note the Learned Model's ability to mimic the non-linear bias curve of the Linear Counting regime.

- Asymptotic Regime ($N > 10^5$): For large cardinalities, the model converges to the standard error rate of the raw HLL estimator, confirming that the residual correction term δ correctly decays to zero as the bias disappears.

5.3 Space and Time Complexity

A key advantage of the learned approach is architectural simplicity. Standard HLL++ implementations require storing empirical lookup tables and implementing interpolation logic between the values. In contrast, our MLP model requires $\approx$ 800 KB of weights and executes a fixed number of matrix multiplications ($O(1)$) regardless of the stream size. This makes it a viable lightweight alternative for systems where code complexity and maintenance are concerns.

5.4 Real-World Data

To verify that our model generalizes beyond synthetic data, we evaluated it on the Complete Works of Shakespeare dataset [16], consisting of 25,752 distinct words. Table 1 compares the results of our Residual Learned Model against the standard HLL implementation. Despite being trained exclusively on uniform random integers, the learned model successfully generalized to the natural language distribution, achieving a relative error of 0.04%. This represents an improvement over the standard analytical estimator (0.07%) on this dataset. This result shows that the model has learned the statistical properties of the HyperLogLog registers rather than overfitting to the training data generation process.

Table 1: Counting Distinct Words in Shakespeare's Complete Works

Method	**Estimate**	**Relative Error**
True Cardinality	25,752	0%
HyperLogLog	25,734	0.0682%
Learned HLL	25,741	0.0391%

6 Conclusion

In this work, we presented a novel hybrid approach to cardinality estimation that augments the standard HyperLogLog algorithm with a lightweight neural network. By framing the bias correction problem as a residual learning task, we demonstrated that a simple Multilayer Perceptron (MLP) can effectively learn the non-linear error patterns inherent in probabilistic counting, replacing the lookup tables and interpolation logic used in industry-standard implementations like Google HLL++.

Our experimental results show that this learned approach provides a unified correction function across all cardinality regimes. On synthetic data, the model successfully reproduced the theoretical lower bounds of Linear Counting without explicit programming. On real-world natural language data (Shakespeare corpus), the model generalized well, achieving a relative error of $\approx 0.04\%$, outperforming the standard HLL estimator while maintaining constant $O(1)$ memory usage.

These findings suggest that "Learned Algorithms", specifically replacing heuristic components of classical data structures with learned models, offer a promising direction for systems optimization.

References

[1] Hazar Harmouch and Felix Naumann. Cardinality estimation: an experimental survey. *Proc. VLDB Endow.*, 11(4):499–512, December 2017.

[2] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. *Discrete Mathematics & Theoretical Computer Science*, DMTCS Proceedings vol. AH, 2007 Conference on Analysis of Algorithms (AofA 07), Jan 2007.

[3] Stefan Heule, Marc Nunkesser, and Alexander Hall. Hyperloglog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, page 683–692, New York, NY, USA, 2013. Association for Computing Machinery.

[4] Kyu-Young Whang, Brad T. Vander-Zanden, and Howard M. Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Trans. Database Syst.*, 15(2):208–229, June 1990.

[5] Otmar Ertl. New cardinality estimation algorithms for hyperloglog sketches. *CoRR*, abs/1702.01284, 2017.

[6] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. *CoRR*, abs/1712.01208, 2017.

[7] Renzhi Wu, Bolin Ding, Xu Chu, Zhewei Wei, Xiening Dai, Tao Guan, and Jingren Zhou. Learning to be a statistician: learned estimator for number of distinct values. *Proceedings of the VLDB Endowment*, 15(2):272–284, October 2021.

[8] Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

[9] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[10] Andrew L. Maas. Rectifier nonlinearities improve neural network acoustic models. 2013.

[11] George Marsaglia. Xorshift rngs. *Journal of Statistical Software*, 8, 01 2003.

[12] Peter J. Huber. Robust Estimation of a Location Parameter. *The Annals of Mathematical Statistics*, 35(1):73 – 101, 1964.

[13] Ilya Loshchilov and Frank Hutter. Fixing weight decay regularization in adam. *CoRR*, abs/1711.05101, 2017.

[14] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. *CoRR*, abs/1912.01703, 2019.

[15] AdRoll. python-hll: A python implementation of hyperloglog, 2019. Accessed: 2025-01-14.

[16] Project gutenberg.