Lifting the Curse of Multidimensional Data with Learned Existence Indexes

Stephen Macke[1]*, **Alex Beutel**[2], **Tim Kraska**[3]*,
Maheswaran Sathiamoorthy[2], **Derek Zhiyuan Cheng**[2], **Ed H. Chi**[2]
[1]UIUC, [2]Google Brain, [3]MIT
[1]smacke@illinois.edu
[2]{alexbeutel,nlogn,zcheng,edchi}@google.com
[3]kraska@mit.edu

Abstract

Bloom filters are a form of existence index which offer compressed representations of sets at the cost of some false positives. Recent work has introduced the notion of *learned Bloom filters*, which can leverage salient differences between in-index elements and out-of-index queries to offer further compression. In this extended abstract, we show that learned filters can handle multidimensional data particularly well, giving space savings of up to **72%** in our experiments. We show how to maximize performance by leveraging three key optimizations: separate modeling of high-cardinality versus low-cardinality dimensions, Bloom filter sandwiching, and robust learning.

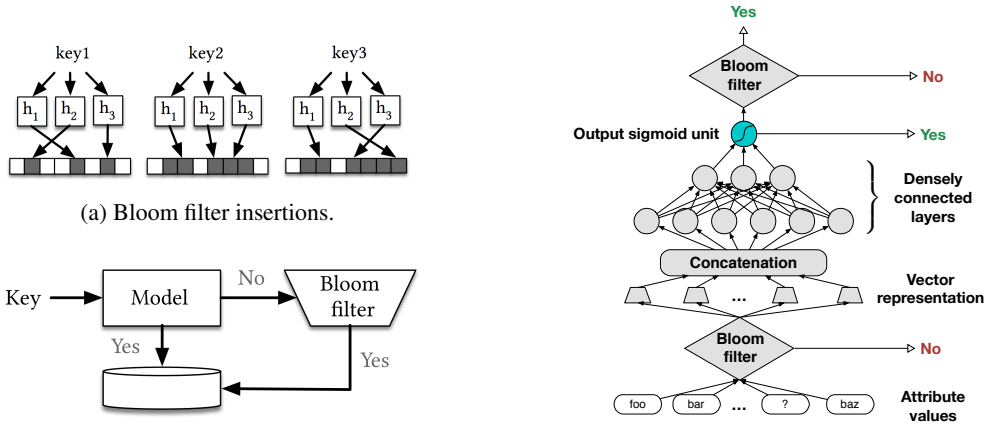
1 Introduction

Bloom filters are widely used as existence indices in many applications [3]. By allowing some false positives, their space requirements depend only on the number of inserted keys (and are independent of the sizes of the individual keys). Despite their advantages, Bloom filters are unable to leverage differences between the set of in-index keys and the distribution of negative queries, and recent work [6] has proposed *learned Bloom filters* to model latent separations between the two. In this work, we expand on these ideas and show that learned filters are effective for indexing multidimensional data (k-tuples) when there exists some co-occurrence structure between in-index k-tuples that are different from out-of-index tuples. We explore a new kind of multidimensional Bloom index that exploits this structure, and our approach can offer significant space savings over traditional Bloom filters. We also give new methods to quantify the space savings that properly capture the asymmetry intrinsic to Bloom filters (i.e. allowing false positives, but not false negatives).

Multidimensional Data. Bloom filters are commonly used to index data with multiple attributes. Furthermore, any application using multiple Bloom filters to index different data into semantically different sets can equivalently use a single, monolithic filter over 2-dimensional data: instead of testing whether Bloom filter i contains item x, we check whether our large filter contains (i, x). For a concrete example, the Windows implementation of the Git SCM uses Bloom filters to support filtered searches over the Git commit graph [2]. Specifically, each commit is associated with a Bloom filter into which all the file names for files added, deleted, or otherwise modified in the commit are inserted. Such an application is semantically equivalent to using a single filter over two dimensions – instead of querying the Bloom filter for commit `ad7ad3b` for file `file.txt`, one would instead query a single monolithic filter for the 2-tuple `(ad7ad3b, file.txt)`.

*Work done while visiting Google

32nd Conference on Neural Information Processing Systems (NIPS 2018), Montréal, Canada.



(a) Bloom filter insertions.

(b) Learned Bloom filter lookups.

(c) Architecture for learned filter over multidimensional data.

Figure 1: Traditional and learned Bloom filters.

We observe that querying over subsets of attributes is often important in many applications; going with our Git example, if we want to check whether any commit modified file `name`, we can do so by querying for `(?, name)` where "?" represents a placeholder that can be matched by any value. In this work, we thus allow *wildcard queries*, in which only a subset of tuple entries are specified. To facilitate such queries, traditional Bloom filters must index tuples like $(a, b, ?)$ as first-class items.

We posit (and demonstrate) that learned filters can be much more space-efficient when allowing wildcard queries, since we observe in practice that co-occurrence structure is present even when only considering subsets of attributes. Even before taking into account all of the extra keys resulting from introduction of wildcards, the number of distinct combinations of attribute values tends can grow exponentially as we increase the number of indexed dimensions. Learned filters have the ability to combat this by leveraging the co-occurrence structure present in the in-index data.

2 Learning Filters

Background. The idea behind learned Bloom filters, as originally described in [6], is illustrated in Figures 1a and 1b. Bloom filter insertions are facilitated by setting h bits in a bit array, whose positions correspond to the output of h independent random hash functions. To determine whether a key is considered in-index, all h hash function outputs must hash to set bits. In this way, false positives are possible, but false negatives are not. Learned Bloom filters model Bloom filter lookups as a classification problem. To train a learned Bloom filter, in-index items, $\mathcal{K}$, are used as positive training examples, and negative training examples are drawn from a negative query distribution $\mathcal{D}_{\overline{\mathcal{K}}}$. Because the model may have false negatives, the set of all false negatives are inserted into a post-filter ("fixup" filter) that is checked whenever the model gives a negative prediction. Concretely, for a classifier f with prediction threshold τ, all $k \in \mathcal{K}$ for which $f(k) < \tau$ (the set of which is denoted as $\mathcal{K}_{<\tau}$) are inserted into the spillover filter. This backup filter is then checked whenever f gives an output below τ in order to eliminate the possibility of false negatives. For a classifier with false positive rate $F_p^{(1)}$ and a backup traditional Bloom filter with false positive rate $F_p^{(2)}$, the overall false positive rate is of the learned Bloom filter is given by $F_p^{(1)} + (1 - F_p^{(1)})F_p^{(2)}$. When $F_p^{(1)}$ is small, this is approximately $\approx F_p^{(1)} + F_p^{(2)}$. For a detailed discussion of how classifier performance relates to learned Bloom filter size, please see [6] or [7].

Modeling Multidimensional Data. Figure 1c depicts our model architecture for handling multidimensional inputs. In this work, we consider k-tuples of strings. Each string attribute value is first converted into an embedding vector (we defer specific discussion to the next section). Furthermore, each "wildcard" placeholder is assigned a separate embedding vector per dimension. To model any co-occurrence structure between the attribute values, these vectors are concatenated and fed through

a dense layer, whose output is then converted to a logit by a sigmoid head. We train the DNN and the embeddings using Adam [5] on positive examples that represent the in-index elements and negative examples synthesized from queries for out-of-index data.

3 Challenges and Optimizations

We identify three key challenges when learning Bloom filters over multidimensional data:

- **Challenge 1: Tradeoff between model size and model performance.**
- **Challenge 2: Suboptimal performance at low false positive rates.**
- **Challenge 3: Noisy in-index data obscures overall co-occurrence patterns.**

We introduce three key optimizations: *(1) modeling high-cardinality attributes with RNNs*, *(2) leveraging the sandwiching technique introduced in [8]*, and *(3) robust learning via ℓ_1-regularized shift parameters* in order to address each of these respective challenges. We now give a detailed description of each optimization.

Optimization 1: Model high-cardinality attributes with RNNs. Using direct embedding lookups for high-cardinality attributes creates many model parameters, and in the worst case results in learned Bloom filters that greatly exceed traditional Bloom filters in size. To cope with this, we convert attribute values for high-cardinality attributes (say, > 500 distinct values) into vectors using character-level recurrent neural networks (RNNs) with gated recurrent units (GRUs) [4]. We opt for direct embeddings for the low-cardinality attributes when converting their corresponding attribute values to vectorized form, as the benefit in learned Bloom filter size due to the improvement in the quality of the model predictions outweighs the cost of extra space used due to the increase in model parameters (compared to RNNs).

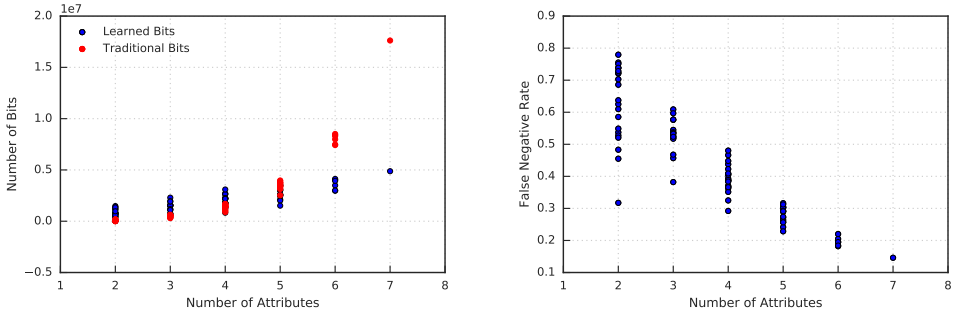
Optimization 2: Choose better classifier cutoffs with sandwiching. To enforce a low target false positive rate *FPR*, it can be necessary to choose a very large classifier cutoff τ. Although there exist strategies for improving performance at low false positive rates, we opt to sidestep the issue altogether by employing a strategy for learned Bloom filters called *sandwiching* [8]. In brief, this strategy introduces an additional traditional Bloom filter in the learned Bloom filter architecture depicted in Figure 1b as a *prefilter*, into which all in-index keys in $\mathcal{K}$ are inserted. Thus, a query must be found in the prefilter to be considered positive; otherwise, it is immediately known to be out-of-index. Although all of $\mathcal{K}$ is inserted into the prefilter, it need not occupy many bits as it can be very porous, having a very high false positive rate compared to the target false positive rate *FPR*.

When leveraging sandwiching, we are free to choose the classifier cutoff τ for the model portion of the learned Bloom filter however we wish. A natural question, then, is how to choose this threshold. The following theorem shows that the best cutoff for a sandwiched filter with nonempty prefilter maximizes the KL divergence between the classifier's true positive and false positive rates.

Theorem 1. *Suppose one has surrounded two learned models with traditional Bloom filters to create sandwiched Bloom filters (with same FPR), and further assume that each sandwiched filter uses the optimal number of bits and leverages a nonempty prefilter to do so. Let X_{T_p} and X_{F_p} be Bernoulli random variables with $\mathbb{E}\left[X_{T_p}\right] = T_p$ and $\mathbb{E}\left[X_{F_p}\right] = F_p$, where T_p is the fraction of in-index items correctly classified by the learned model, and F_p is the learned model's false positive rate (expected proportion of out-of-index queries predicted as in-index). Then the sandwiched filter with the higher value of KL$(X_{T_p}||X_{F_p})$ uses fewer bits for the traditional prefilter and postfilter components.*

Proof. Please see Appendix A. □

Optimization 3: Robust learning with ℓ_1-regularized shift parameters. Our final optimization attempts to address noise present in the in-index data. To do so, we borrow an idea from robust learning, whereby each positive training example k is associated with a *shift parameter* γ_k. The vector of shift parameters $\vec{\gamma}$ is ℓ_1-regularized to encourage sparsity. Although originally introduced in the context of linear regression [9] and logistic regression [10], we found that the technique works well in deep models too. In brief, the output of the sigmoid head $g(z_k) = \frac{1}{1+\exp(-z_k)}$ is shifted as $g(z_k + \gamma_k) = \frac{1}{1+\exp(-z_k-\gamma_k)}$. In our experiments on a dataset of airline flights [1], this technique reduced the number of bits required for a learned Bloom filter by 15%.



(a) Effect of # attributes on bits needed

(b) Effect of # attributes on model FNR

Figure 2: Effect of varying number of attributes indexed. Each point corresponds to a learned Bloom filter that achieves 1% FPR for a combination of indexed attributes, across all subsets of two or more indexed attributes.

4 Experimental Results

Production Recommender System Logs. We consider a dataset of 70000 subsampled impressions from the logs of a major production recommendation system, where each impression consists of 7 attributes. Although the number of rows (i.e. impressions) is relatively small, by allowing wildcard-style queries, we observe 1.8 million distinct co-occurring attribute values, all of which must be indexed. Since the false positive rate of learned Bloom filters is sensitive to the distribution of incoming negative queries [6, 7], for each set of indexed attributes, we synthesized an out-of-index query distribution. This distribution was generated by joining random pairs of non co-occurring observed attribute values, uniformly across pairs of indexed attributes (queries with unobserved attribute values will be eliminated during embedding lookups, with high probability), although one could also use query logs if available. Figure 2 shows the result of indexing every subset of attributes, using both a traditional filter and a learned filter. Without RNNs (not shown), the learned model uses more bits than a traditional filter. Leveraging sandwiching reduces the number of bits needed by about 10%. Overall, we achieve greater space savings with more indexed attributes, achieving space reductions of **72%** over traditional Bloom filters when indexing all 7 attributes.

Flights. We divided 7 million rows from year 2008 into contiguous pages (following the order in which we obtained the data) of 10000 rows each, and index all pairs of `(Airport, Page ID)`. We join known airports with random page IDs for a query negative distribution, achieve space reductions of **25%** over traditional bloom filters. By incorporating ℓ_1-regularized shift parameters, we increase the space reductions to **35%**.

5 Conclusions

Based on our experimental findings, learned Bloom filters show promising performance when indexing multidimensional data while allowing flexible queries with wildcards. Each of our key optimizations: *separate modeling of high-cardinality attributes with RNNs*, *sandwiching with optimal classifier cutoffs*, and *robust learning with shift parameters* addresses a key issue when learning Bloom filters over multidimensional data.

A Proof of Theorem 1

Theorem 1. *Suppose one has surrounded two learned models with traditional Bloom filters to create sandwiched Bloom filters (with same FPR), and further assume that each sandwiched filter uses the optimal number of bits and leverages a nonempty prefilter to do so. Let X_{T_p} and X_{F_p} be Bernoulli random variables with $\mathbb{E}\left[X_{T_p}\right] = T_p$ and $\mathbb{E}\left[X_{F_p}\right] = F_p$, where T_p is the fraction of in-index items correctly classified by the learned model, and F_p is the learned model's false positive rate (expected proportion of out-of-index queries predicted as in-index). Then the sandwiched filter with the higher value of KL$(X_{T_p}||X_{F_p})$ uses fewer bits for the traditional prefilter and postfilter components.*

Proof. Following the analysis in [8], the overall false positive rate of a sandwiched filter with b bits per key and a learned model with false positive rate F_p and false negative rate F_n is given by

$$\alpha^{b-b_2}\left(F_p + (1-F_p)\alpha^{b_2/F_n}\right)$$

where the second filter has b_2 bits per key (so that the first filter has $b - b_2$ bits per key), and $\alpha \approx 0.6185$ is chosen assuming each traditional filter uses an optimal number of hash functions (in the sense of minimizing their respective false positive rates). In more detail, the first filter has a false positive rate of α^{b-b_2} and the second filter has a false positive rate of α^{b_2/F_n}, where the "effective bits per key" b_2/F_n in the second traditional filter is scaled up because only the model false negatives need be inserted.

Thus, the overall bits per key in an optimal sandwiched filter is given by

$$b = \log_\alpha(\textit{FPR}) + b2 - \log_\alpha\left(F_p + (1-F_p)\alpha^{b_2/F_n}\right)$$

where *FPR* gives the desired false positive rate of the whole sandwiched setup. Mitzenmacher showed in [8] that the optimal value of b_2 is given by

$$b_2 = F_n \log_\alpha\left(\frac{F_p F_n}{(1-F_p)(1-F_n)}\right)$$

independently of the overall bits per key b. Therefore, we can attempt to choose a classifier cutoff which leads to model false positive and false negative rates F_p and F_n that minimize the overall bits per key, equivalent to minimizing

$$b_2 - \log_\alpha\left(F_p + (1-F_p)\alpha^{b_2/F_n}\right)$$

independently of the target false positive rate *FPR*. Plugging in the optimal value for b_2 given above, we are minimizing

$$\begin{aligned}
& b_2 - \log_\alpha\left(F_p + (1-F_p)\alpha^{b_2/F_n}\right) \\
=\ & F_n \log_\alpha\left(\frac{F_p F_n}{(1-F_p)(1-F_n)}\right) - \log_\alpha \frac{F_p}{1-F_n} \\
=\ & F_n \log_\alpha \frac{F_n}{1-F_p} + F_n \log_\alpha \frac{F_p}{1-F_n} - \log_\alpha \frac{F_p}{1-F_n} \\
=\ & F_n \log_\alpha \frac{F_n}{1-F_p} + (1-F_n) \log_\alpha \frac{1-F_n}{F_p} \\
=\ & \frac{1}{\log \alpha}\text{KL}(F_n||1-F_p) \\
=\ & \frac{1}{\log \alpha}\text{KL}(1-F_n||F_p) \\
=\ & \frac{1}{\log \alpha}\text{KL}(T_p||F_p)
\end{aligned}$$

where $T_p = 1 - F_n$ is the model's true positive rate. Because $\log \alpha < 0$, our final result is that, in order to minimize the bits per key b of the sandwiched filter, we should choose the model's classification threshold in order to maximize the KL divergence between the Bernoulli distributions of true positives and false positives. □

References

[1] Flight Records. http://stat-computing.org/dataexpo/2009/the-data.html, 2009. Accessed: July 2018.

[2] Supercharging the Git Commit Graph IV: Bloom Filters. https://blogs.msdn.microsoft.com/devops/2018/07/16/super-charging-the-git-commit-graph-iv-bloom-filters/, 2018. Accessed: July 2018.

[3] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet mathematics*, 1(4):485–509, 2004.

[4] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[5] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[6] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pages 489–504. ACM, 2018.

[7] M. Mitzenmacher. A model for learned bloom filters and related structures. *arXiv preprint arXiv:1802.00884*, 2018.

[8] M. Mitzenmacher. Optimizing learned bloom filters by sandwiching. *arXiv preprint arXiv:1803.01474*, 2018.

[9] Y. She and A. B. Owen. Outlier detection using nonconvex penalized regression. *Journal of the American Statistical Association*, 106(494):626–639, 2011.

[10] J. Tibshirani and C. D. Manning. Robust logistic regression using shift parameters (long version). *arXiv preprint arXiv:1305.4987*, 2013.