# Partitioned Learned Bloom Filters

**Kapil Vaidya**
CSAIL
MIT
kapilv@mit.edu

**Eric Knorr**
School of Engineering and Applied Sciences
Harvard University
eric.r.knorr@gmail.com

**Tim Kraska**
CSAIL
MIT
kraska@mit.edu

**Michael Mitzenmacher**
School of Engineering and Applied Sciences
Harvard University
michaelm@eecs.harvard.edu

## Abstract

Learned Bloom filters enhance standard Bloom filters by using a learned model for the represented data set. However, a learned Bloom filter may under-utilize the model by not taking full advantage of the output. The learned Bloom filter uses the output score by simply applying a threshold, with elements above the threshold being interpreted as positives, and elements below the threshold subject to further analysis independent of the output score (using a smaller backup Bloom filter to prevent false negatives). While recent work has suggested additional heuristic approaches to take better advantage of the score, the results are only heuristic. Here, we instead frame the problem of optimal model utilization as an optimization problem. We show that the optimization problem can be effectively solved efficiently, yielding an improved *partitioned learned Bloom filter*, which partitions the score space and utilizes separate backup Bloom filters for each region. Experimental results from both simulated and real-world datasets show significant performance improvements from our optimization approach over both the original learned Bloom filter constructions and previously proposed heuristic improvements.

## 1 Introduction

Bloom filters are space-efficient probabilistic data structures that are used to test whether an element is a member of a set [Bloom (1970)]. A Bloom filter may allow false positives, but will not give false negative matches, which makes them suitable for numerous applications in networks, databases and other systems areas. Indeed, there are many thousands of papers describing applications of Bloom filters [Dayan et al. (2018), Dillinger and Manolios (2004), Broder and Mitzenmacher (2003)]. For standard Bloom filters, there are known theoretical lower bounds on the space used [Pagh et al. (2005)]. However, these lower bounds assume the Bloom filter could store any possible set. If the data set or the membership queries have specific structure, it may be possible to beat the lower bounds in practice [Mitzenmacher (2002), Bruck et al. (2006), Mitzenmacher et al. (2020)]. In particular, Kraska et al. (2018) and Mitzenmacher (2018) propose using machine learning models to reduce the space further, by using the learned model to provide a suitable pre-filter for the membership queries. Rae et al. (2019) propose a neural Bloom Filter that learns to write to memory using a distributed write scheme and achieves compression gains over the classical Bloom filter.

Here, we focus on the work of Kraska et al. (2018), which studies how standard index structures, including Bloom filters, can be improved using machine learning models. Given an input the model outputs a score which is supposed to represent the confidence of the input being in the set. Thus, the inputs in the set (keys) should have a higher score value compared to the rest of the inputs (non-keys). This model is used as a pre-filter, where inputs with score values above a threshold are directly

classified as being in the set. A smaller backup Bloom filter is used for rest of the inputs, which maintains the property that there are no false negatives. The threshold value is essentially used to partition the space of scores into two regions, with one region above the the threshold and one region below. The input is processed differently depending on which region its score falls. With a sufficiently accurate model, the size of the backup Bloom filter can be reduced significantly over the size of a standard Bloom filter while maintaining overall accuracy. The later work in Mitzenmacher (2018) grows out of Kraska et al. (2018) by providing a formal analysis for the learned Bloom filter structure and proposing some improvements, based on "sandwiching" the pre-filter between two Bloom filters (instead of just a backup).

Intuitively, we might be able to do even better by partitioning the score space into more than two regions, and optimizing the processing for each region. That is, simply using a single threshold underutilizes the model, by only comparing the score to a single threshold value. We are not the first to notice this. The work Dai and Shrivastava (2019) suggests using multiple thresholds to divide the space of scores into multiple regions, with a different backup Bloom filter for each score region. The parameters for each of the backup Bloom filters are chosen to improve the overall tradeoff between the false positives and size. However, Dai and Shrivastava (2019) only suggests heuristics for how to divide up the score space and how to choose the false positive rate (and corresponding Bloom filter size) for each region of the score space.

In this work, we show how to frame the overall problem as an optimization problem, which can significantly outperform the heuristic used in Dai and Shrivastava (2019). Moreover, we show that our space savings is linearly proportional to the KL Divergence of the key and non-key score distributions. We present a dynamic programming algorithm to find optimal parameters (up to the discretization used for the dynamic programming) and demonstrate performance improvements over a synthetic dataset and two real world datasets: URL's and EMBER. We note that Dai and Shrivastava (2019) refers to their structure as an adaptive learned Bloom filter, but *adaptive* has been in multiple papers in multiple ways in the context of Bloom filters; we therefore refer to our approach as a *partitioned learned Bloom filter* (PLBF). Experimental results show the amount of space saved by PLBF is 2x more than sandwiching approach and 1.7x more than the adaptive LBF heuristics.

## 2 Review

We first start by reviewing the standard Bloom filters and some variants of the learned Bloom filter.

### 2.1 Standard Bloom Filter

A Bloom filter representing a set $S = \{x_1, x_2, ..., x_n\}$ of $n$ keys is represented by an array of $m$ bits and uses $k$ independent hash functions $\{h_1, h_2, ...h_k\}$ with the range of each $h_i$ being integer values between 0 and $m-1$. We assume the hash function are fully random. Initially all $m$ bits are 0. For every key ($x \in S$), array bits $h_i(x)$ are set to 1 for all $i \in \{1, 2, ...k\}$.

A membership query for $y$ returns true (that is, $y \in S$) if $h_i(y) = 1$ for all $i \in \{1, 2, ...k\}$ and false otherwise. This ensures that the Bloom filter has no false negatives but non-keys ($y \notin S$) might return true resulting in a false positive. This false positive rate depends on the space $m$ used by the Bloom Filter. The expected false positive rate is given in Eq.1 ; if one uses the optimal number of hash functions $k = m \ln 2/n$, the false positive rate is as in Eq.2.[1]

$$E[FPR] = \left(1 - \left(1 - \tfrac{1}{m}\right)^{kn}\right)^k \tag{1}$$

$$E[FPR] = \left(\tfrac{1}{2}\right)^{\frac{m}{n} * \ln 2} \tag{2}$$

### 2.2 Learned Bloom Filter

We are given a set of keys $S = \{x_1, x_2, .., x_n\}$ for which to build a Bloom filter. We are also given a sample of the non-keys $Q$ for training. The learned model is then trained on $S \cup Q$ for a binary

[1] We note that these expressions are just very accurate approximations; see Broder and Mitzenmacher (2003); Bose et al. (2008) for further details.

classification task and produces a score $s(x) \in [0, 1]$. This score $s(x)$ can be viewed as an estimate of the probability that the element $x$ is in the set $S$, so a key ($x \in S$) would ideally have a higher score value than the non-keys ($x \in Q$).

As discussed above, Kraska et al. (2018) set a threshold $t$ and inputs satisfying $s(x) > t$ are classified as a key. This process might result in false negatives as some keys ($x \in S$) might have score values below the threshold. A backup Bloom filter is built for just the keys in $S$ satisfying $s(x) \leq t$; if few keys have scores below the threshold, the backup Bloom filter can be much smaller that a Bloom filter for the original set. The threshold value divides the score space into two regions ($s(x) > t$ and $s(x) \leq t$), with inputs being subjected to different processing based on which region their score values fall. This design is represented in Fig.1(A).

Mitzenmacher (2018) proposes using another Bloom filter *before* the learned model along with a backup Bloom Filter. As the learned Model is used between two Bloom Filters as shown in Fig.1(B), this is referred to as the 'sandwiching' approach. They also provide the analysis of the optimal false positive rate for both learned and sandwiched Bloom filters. Interestingly, the sandwiching approach and analysis can be seen as a special case of our approach and analysis, as we describe later in subsection 3.4.1.

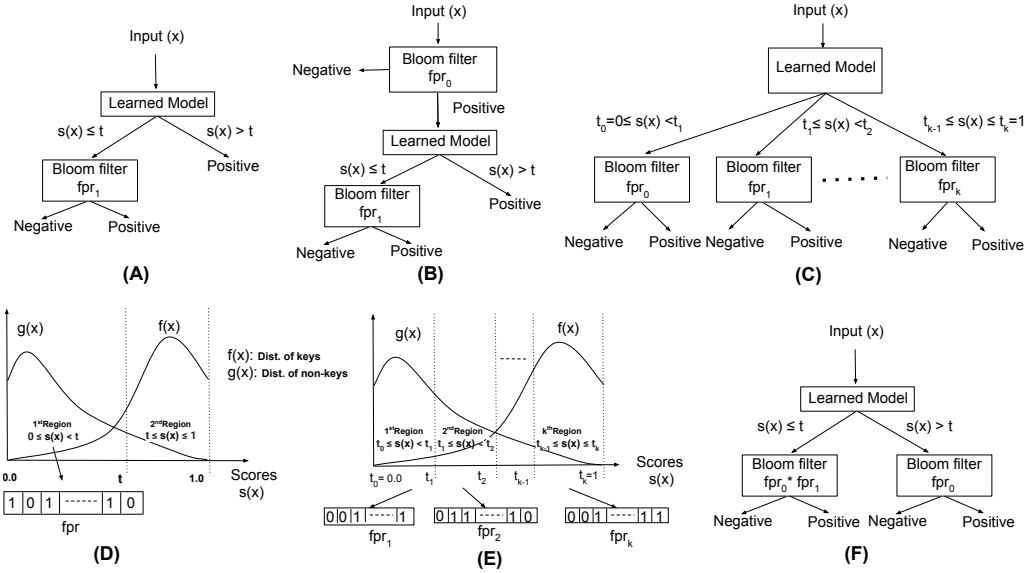


Figure 1: (A),(B),(C) represent the original LBF, LBF with sandwiching, and PLBF designs, respectively. Each region in (C) is defined by score boundaries $t_i, t_{i+1}$ and a false positive rate $fpr_i$ of the Bloom Filter used for that region. (D),(E) show the LBF and PLBF with score space distributions. (F) represents a PLBF design equivalent to the sandwiching approach used in 3.4.1.

# 3 Partitioned Learned Bloom Filter (PLBF)

## 3.1 Design

In the original learned Bloom filter, the scores of the inputs are divided into two regions based on a threshold. The information in the score value is never used beyond comparing it to the threshold. As the score value represents the probability of the input being a key, we can utilize the score better if we segment the scores space into more regions using multiple thresholds, as shown in Fig.1(C), where we use separate backup Bloom filters for each region. We can choose different target false positive rates for each region[2]. The parameters associated with each region are its threshold boundaries and its false positive rate. Setting good values for these parameters is crucial for performance. Our aim

[2]The different false positive rates for each region can be materialized in variety of ways. We can either choose a separate Bloom filter for each region or have a common Bloom filter with varying number of hash functions per region.

is to analyze the performance of the learned Bloom filter with respect to these parameters, and find methods to determine optimal or near-optimal parameters.

Interestingly, we find in our formulation that the resulting parameters correspond to quite natural quantities. Specifically, the optimal false positive rate of a region is proportional to the ratio of key and non-key density in the region. Further, the optimal threshold values turn out to be those maximizing the KL divergence between the key and non-key distributions of the regions.

In what follows, we formulate the problem as an optimization problem in subsection 3.2. In subsection 3.3.1, we find the optimal solution of a relaxed problem which helps us gain some insight into the general problem. We propose a solution for the general problem in subsection 3.3.3.

### 3.2 General Optimization Formulation

To formulate the overall problem as an optimization problem, we consider the variant which minimizes the space used by the Bloom filters in PLBF in order to achieve an overall a target false positive rate ($FPR$). Here we are assuming the learned model is given. We could have similarly framed it as minimizing the false positive rate given a fixed amount of space.

We assume normalized score values in $[0, 1]$ for convenience. We have region boundaries given by $t_i$ values $0 = t_0 \leq t_1 \leq t_2 ..... t_{k-1} \leq t_k = 1$, with score values between $[t_{i-1}, t_i]$ falling into the $i^{th}$ region. We assume the target number of regions $k$ is given. The $i^{th}$ region has false positive rate $fpr_i$. Let $f$ be the probability distribution of the keys in the score space and $F$ the CDF corresponding to $f$. Similarly, let $g$ and $G$ be the PDF and CDF distribution of non-keys, respectively. Fig.1(D) represents a standard learned Bloom filter with this framwork, while Fig.1$(E)$ represents $k$ score regions and the parameters for each region. The following optimization problem finds the thresholds $t_i$ and the false positive rates $fpr_i$:

$$\min_{t_{i=1\ldots k-1}, fpr_{i=1\ldots k}} \quad \sum_{i=1}^{k} |S| * (F(t_i) - F(t_{i-1})) * \log_2 \left(\frac{1}{fpr_i}\right) \tag{3}$$

$$\text{constraints} \quad \sum_{i=1}^{k} (G(t_i) - G(t_{i-1})) * fpr_i \leq FPR \tag{4}$$

$$fpr_i \leq 1 \ \ , i = 1 \ldots k \tag{5}$$

$$(t_i - t_{i-1}) \geq 0 \ \ , i = 1 \ldots k \ ; \ t_0 = 0; t_k = 1 \tag{6}$$

The minimized term (Eq.3) represents the total size of the backup Bloom filters, obtained by summing the individual backup Bloom filter sizes [3]. The first constraint (Eq.4) ensures that the overall false positive rate stays below the target FPR, as the overall false positive rate is obtained by summing the appropriately weighted rates of each region. The next constraint (Eq.5) ensures that the false positive rates of each class stay below one. The last two constraints (Eq.6) ensure threshold values are increasing and cover the interval $[0, 1]$.

### 3.3 Solving the Optimization Problem

#### 3.3.1 Solving a Relaxed Problem

In this subsection, we look at the optimal solution of a relaxed problem, obtained after removing the false positive rate constraints (Eq.5, giving $fpr_i \leq 1$), as shown in Eq.7. This relaxation is useful because it allows us to use the Karush-Kuhn-Tucker (KKT) conditions to obtain an exact solution in terms of the $t_i$ values, which we used to design algorithms for finding near-optimal solutions. Throughout this section, we assume the the relaxed problem yields a solution for the original problem; we return to this issue in subsection 3.3.3.

[3] The size of a Bloom filter is proportional to $|S| * \log_2(1/fpr)$, where $S$ is the set it represents, and $fpr$ is the false positive rate it achieves. See e.g. Mitzenmacher (2018) for related discussion.

$$\begin{aligned} \min_{t_{i=1\ldots k-1}, fpr_{i=1\ldots k}} \quad & \sum_{i=1}^{k} |S| * (F(t_i) - F(t_{i-1})) * \log_2\left(\frac{1}{fpr_i}\right) \\ \text{constraints} \quad & \sum_{i=1}^{k} (G(t_i) - G(t_{i-1})) * fpr_i \leq FPR \\ & (t_i - t_{i-1}) \geq 0 \;\; i = 1...k;\; t_0 = 0;\; t_k = 1 \end{aligned} \tag{7}$$

The optimal $fpr_i$ values obtained by using the KKT conditions yield Eq.8 (as shown in the appendix. A), giving the exact solution in terms of $t_i$'s.

$$\frac{fpr_i}{FPR} = \frac{F(t_i) - F(t_{i-1})}{G(t_i) - G(t_{i-1})} \tag{8}$$

The numerator $F(t_i) - F(t_{i-1})$ is the key density in the $i^{th}$ region and the denominator $G(t_i) - G(t_{i-1})$ is the non-key density in the $i^{th}$ region. The optimal $fpr_i$ is proportional to the ratio of key density to non-key density in a region. This is intuitive as a region with very high key density should allow most inputs to pass as most of them are keys, allowing a high false positive rate for a backup Bloom filter. Similarly, a region with low key density requires a Bloom filter with low false positive rate to prevent non-keys from being allowed.

Since, we have optimal $fpr_i$ in terms of $t_i$, we can replace $fpr_i$ in the original problem to get a problem only in terms of $t_i$. Eq.9 shows the rearrangement of the minimization term after substitution.

$$\begin{aligned} \text{Minimization Term} &= \sum_{i=1}^{k} |S| * (F(t_i) - F(t_{i-1})) * \log_2\left(\frac{G(t_i) - G(t_{i-1})}{(F(t_i) - F(t_{i-1})) * FPR}\right) \\ &= \sum_{i=1}^{k} |S| * (F(t_i) - F(t_{i-1})) * \log_2\left(\frac{1}{FPR}\right) - |S| * D_{KL}\left(\hat{f(\mathbf{t})}, \hat{g(\mathbf{t})}\right) \end{aligned} \tag{9}$$

Again, the term being minimized (Eq.9) represents the total space occupied by the backup Bloom filters; the total space used by our method is the sum of backup Bloom filter space and space occupied by the learned model.

$$|S| * \log_2\left(\tfrac{1}{FPR}\right) - |S| * D_{KL}\left(\hat{f(\mathbf{t})}, \hat{g(\mathbf{t})}\right) + \text{Size Of Learned Model} \tag{10}$$

The space occupied by a standard Bloom filter is $|S| * \log_2(1/FPR)$. Thus, the space saved by PLBF in comparison to the standard Bloom filter is:

$$|S| * D_{KL}\left(\hat{f(\mathbf{t})}, \hat{g(\mathbf{t})}\right) - \text{Size Of Learned Model} \tag{11}$$

The space saved by PLBF is therefore linearly proportional to the KL divergence of key and non-key densities($\hat{f(\mathbf{t})}, \hat{g(\mathbf{t})}$) of the regions.

### 3.3.2 Finding the Optimal Thresholds

We have shown that, given a set of thresholds, we can in the relaxed problem find the optimal false positive rates. Here we turn to the question of finding optimal thresholds. We assume that we are given $k$, the number of regions desired. (We consider the importance of choosing $k$ further in our experimental section.) Given our results above, the optimal thresholds correspond to the points that maximize the KL divergence between ($\hat{f(\mathbf{t})}, \hat{g(\mathbf{t})}$). The KL divergence of ($\hat{f(\mathbf{t})}, \hat{g(\mathbf{t})}$) is the sum of the terms $f_i \log_2 \frac{f_i}{g_i}$, one term per region. Note each term depends only on the proportion of keys and non-keys in that region and is otherwise independent of the other regions. This property allows a recursive definition of KL divergence that is suitable for dynamic programming.

We divide the score space $[0, 1]$ into $N$ consecutive small segments; this provides us a discretization of the score space, with larger $N$ more closely approximating the real interval. Given $k$, we can find the set of $k$ optimal thresholds using dynamic programming. Let $DP_{KL}(n, j)$ denote the maximum divergence one can get when you divide the first $n$ segments into $j$ parts. Our optimal divergence

corresponds to $DP_{KL}(N,k)$. The idea behind the algorithm is that the we can recursively define $DP_{KL}(n,j)$ as represented in Eq.12. Here $f', g'$ represent the probability distribution of keys and non-keys in these $N$ segments.

$$DP_{KL}(n,j) = \max\left(DP_{KL}(n-i,j-1) + \left(\sum_{t=i}^{n} f'(t) * \log_2\left(\frac{\sum_{t=i}^{n} f'(t))}{\sum_{t=i}^{n} g'(t)}\right)\right)\right) \tag{12}$$

The time complexity of computing $DP_{KL}(N,k)$ and the optimal thresholds is $\mathcal{O}(N^2k)$. Hence one can increase the value of $N$ to get more precise thresholds at the cost of higher computation time.

#### 3.3.3 The Relaxed Problem and the General Problem

Once we obtain the threshold values that maximize the divergence, we can get the optimal $fpr_i$ values using Eq.8. In many cases the relaxed solution will also be the optimal solution of the general problem. Specifically, if $f(x)/g(x) < 1/FPR$ for all $x$, then $fpr_i < 1$ for all intervals in the relaxed solution. (This is because $f(x)/g(x) < 1/FPR$ implies $FPR \cdot (F(t_{i-1}) - F(t_i))/(G(t_{i-1}) - G(t_i)) < 1$ for al $i$.) In this case, optimal solution of the relaxed problem is also optimal for the general problem. Hence if we are aiming for a sufficiently low false positive rate, the relaxed problem suffices.

In some cases we may first assign false positive rates based on the relaxed problem and Eq.8 but find that $fpr_i \geq 1$ for some regions. For such regions, we can set $fpr_i = 1$, re-solve the relaxed problem with these additional constraints (that is, excluding these regions), and use the result as a solution for the general problem. Some regions might again have a false positive rate above one, so we can repeat the process. The algorithm stops when there is no new region with false positive rate greater than one. Given a set of thresholds, this algorithm is optimal, but this algorithm may not be optimal for the general problem (as the relaxed problem formulation is used to determine the thresholds). However, we expect it will perform very well in most cases (we provide pseudo-code in Appendix.C).

With this enhanced algorithm in mind, we can find additional ways for the relaxed solution to lead us to the optimal solution. Again, the issue is that there might be cases where the $fpr_i$ constraint is violated and here we extend the solution to account for that. In an interval where $fpr_i = 1$, we would want to use no Bloom filter. In this part of the score space, keys occur dramatically more frequently than non-keys, and we may simply choose to accept them as keys from our set. (Indeed, this is what is done in the standard learned Bloom filter.) We may thus improve upon our previous solution by assuming that at most one rightmost region of the score space has no Bloom filter. In particular, if $f$ and $g$ are monotonically increasing and decreasing functions, respectively, then indeed without loss of genearlity the last $k$th region will be the only one with $fpr_k = 1$. Moreover, it is not unreasonable to believe that in practice $f$ and $g$ will be monotonic or nearly so; this simply corresponds to keys being more concentrated on higher scores, and non-keys being more concentrated on lower scores.

The problem of finding the optimal thresholds becomes easier after this assumption as we can safely remove of all the $fpr_i \leq 1$ constraints for $i \neq k$ (and hence we can apply the KKT conditions), and we try all possible boundaries for the $k$th region where $fpr_k \leq 1$. Specifically, as before we divide the score space into $N$ discrete segments. The algorithm then iterates on all the $\mathcal{O}(N)$ possibilities for $t_{k-1}$, setting the false positive rate of the $k$th region to 1, and then uses the previous dynamic programming algorithm of section 3.3.2 for the rest of the segments. The worst case time complexity is then $\mathcal{O}(N^3k)$. After obtaining a set of thresholds form the DP algorithm, we can set the $fpr_i = 1$ for the rightmost region and use Eq.8 for the rest (we provide pseudo-code in the Appendix.B).

### 3.4 Additional Considerations

First, we show how the sandwiching approach is a special case of our design. Next, we will see how the performance varies w.r.t number of regions.

#### 3.4.1 Sandwiching: A Special Case

We show here that the sandwiching approach can actually be interpreted as a special case of our method. In the sandwiching approach, the learned model is sandwiched between two Bloom filters as shown in Fig.1(B). The input first goes through a Bloom filter and the negatives are discarded. The positives are passed through the learned model where based on their score $s(x)$ they are either directly accepted when $s(x) > t$ or passed through another backup Bloom filter when $s(x) \leq t$. In

our setting, we note that the pre-filter in the sandwiching approach can be merged with the backup filters to yield backup filters with a modified false positive rate. Fig.1(F) shows what an equivalent design with modified false positive rates would look like. (Here equivalence means we obtain the same false positive rate with the same bit budget; we do not consider compute time.) Thus, we see that the sandwiching approach can be viewed as a special case of the PLBF with two regions.

However, this also tells us we can make the PLBF more efficient by using sandwiching. Specifically, if we find when constructing a PLBF with $k$ regions that $fpr_i < 1$ for all $i$, we may $fpr_0 = \max_{1 \leq i \leq k} fpr_i$. We may then use an initial Bloom filter with false positive rate $fpr_0$, and change the target false positive rates for all other intervals to $fpr_i / fpr_0$, while keeping the same bit budget. This approach will be somewhat more efficient computationally, as we avoid computing the learned model for some fraction of non-key elements.

#### 3.4.2 Performance against number of regions $k$

Earlier, we saw the space saved by using the PLBF is linearly proportional to the $D_{KL}(\hat{f(\mathbf{t})}, \hat{g(\mathbf{t})})$. If we split any region into two regions, the overall divergence would increase because sum of divergences of the two split regions is always more than the original divergence, as shown in Eq.13. Eq.13 is an application of Jensen's inequality.

$$\left((f_1 + f_2) * \log \tfrac{(f_1+f_2)}{(g_1+g_2)}\right) \leq \left(f_1 * \log \tfrac{f_1}{g_1}\right) + \left(f_2 * \log \tfrac{f_2}{g_2}\right) \tag{13}$$

Increasing the number of regions therefore always improves performance, but the quantity $D_{KL}(\hat{f(\mathbf{t})}, \hat{g(\mathbf{t})})$ has an upper bound $D_{KL}(f, g)$. Hence

$$\lim_{k \to \infty} D_{KL}\left(\hat{f(\mathbf{t})}, \hat{g(\mathbf{t})}\right) = D_{KL}\left(f, g\right) \tag{14}$$

We would hope that in practice a small number of regions $k$ would suffice. This seems to be the the case in our experience; we detail one such experiment (Experiment 4.2.1) in our evaluation.

## 4 Evaluation

The baselines used in our evaluation are the standard Bloom filter [Bloom (1970)], the sandwiching LBF [Mitzenmacher (2018)], and the adaptive learned Bloom filter (AdaBF) [Dai and Shrivastava (2019)]. For the evaluation we use 3 different datasets:

**URLs**: Our first dataset is the URLs dataset, which was also used for evaluation in previous papers [Kraska et al. (2018), Dai and Shrivastava (2019)]. It contains 450176 URLs, of which 103520 (23%) are malicious and the rest 346646 (77%) are benign. We extract 17 features in total from these URL's such as length of host name, use of shortening, counts of various special characters in the URL's, etc.

**EMBER**: Bloom filters are widely used to match file signatures with the virus signature database. Ember (Endgame Malware Benchmark for Research) [Anderson and Roth (2018)] is an open source collection of 1.1 million portable executable file (PE file) sha256 hashes that were scanned by VirusTotal sometime in 2017. Out of the 1.1 million files 400K are malicious, 400K are benign, and 300K are unlabeled files. The features of the files are already included in the dataset.

**Synthetic**: An appealing scenario for our method is when the key density increases and non-key density decreases monotonically with respect to the score value. We simulate this by generating the key and non-key score distribution using Zipfian distributions as in Fig.2(A). Since we directly work on the score distribution the size of the learned model is zero.

As the standard Bloom filter does not use the learned model, to make a fair comparison, we include the size of the learned model with the size of the learned Bloom filter.

### 4.1 Overall Performance

Here, we compare the performance of PLBF w.r.t other baselines. We do this by fixing the target $FPR$ and determining the space used by each method. First, we train the model using 40% percent of the non-key set and the entire key set as training samples. The parameters of each method are then

tuned using this model with the aim of achieving the fixed *target* $FPR$. The rest of the non-keys are used to evaluate the *actual* false positive rate achieved by these methods. We plot this actual false positive rate achieved against the space used by the method.

All the methods can function regardless of what type of model is used. We choose the random forest classifier from sklearn [Pedregosa et al. (2011)] as it yields good accuracy on these datasets. The F1 scores of the learned models used for synthetic, URL's and EMBER were 0.99, 0.97, and 0.85, respectively. We consider the size of the model to be the pickle[4] file size on the disk. This size has been added to the learned Bloom filter baselines as well as our method. Again, no model is used for the synthetic dataset, so the size of the model is zero in this case. We use five regions ($k = 5$) for both PLBF and AdaBF as this is usually enough to achieve good performance as shown in 4.2.1.

The results of the experiment are shown in the Fig.2 along with the distribution of the scores of keys and non-keys $(f, g)$ for each dataset. As we can see from the figure, PLBF gives a better Pareto curve than the other baselines for these datasets. On the synthetic dataset, our performance is much better than the other baselines. The URL's dataset has similar score distribution to the synthetic dataset and we see a similar performance trend. The score distribution for the EMBER dataset indicates that the model here is not as helpful. Due to the limited power of the model, the performance benefit of every method over a standard Bloom filter is much less significant, but still the PLBF offer gains. AdaBF is not consistent w.r.t false positive rate as the performance of its heuristics are not predictable.

The space benefit provided by PLBF versus the standard Bloom filter first increases but converges to a constant time the set size, as given in Eq.11. As the $fpr$ tends to zero, the space saved by PLBF is 1.75x, 2.3x, 1.9x compared to the sandwiching approach and 1.5x, 2x, 1.6x compared to AdaBF on synthetic, URL and EMBER datasets, respectively.

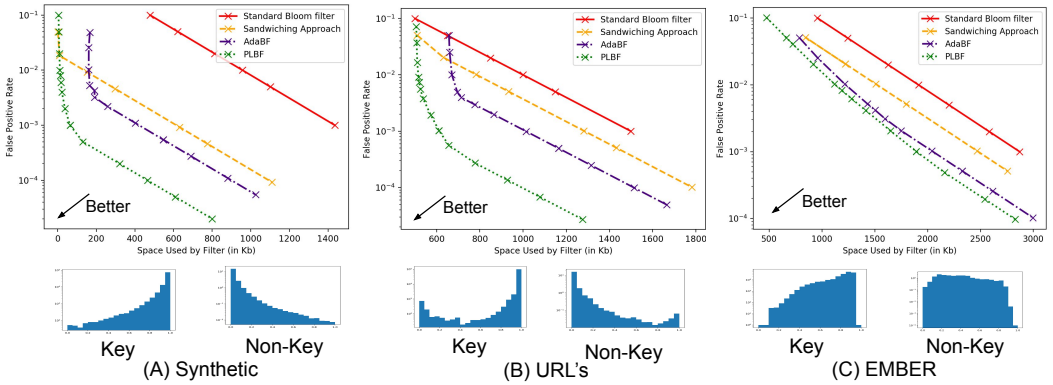


Figure 2: FPR vs Space for the (A) Synthetic (B) URLs (C) EMBER datasets for various baselines along with key and non-key score distributions.

## 4.2 Micro-Benchmarks

Here, we see how the performance of PLBF varies with number of regions(k) and model quality. The following experiments were done on synthetic dataset.

### 4.2.1 Performance and the Number of Regions

Here, we see how the performance (amount of space saved) of our approach varies as we change the number of regions $k$. We have seen that the space savings obtained by a learned model is linearly proportional the divergence of the distributions $D_{KL}(f(\hat{\mathbf{t}}), g(\hat{\mathbf{t}}))$, and this divergence strictly increases with the number of regions but is upper bounded by $D_{KL}(f, g)$. Fig.3(A) shows the space saved as we increase the number of regions $k$ on the synthetic dataset (F1 score=0.99). The red line in the figure represents the upper bound of space savings which corresponds to divergence $D_{KL}(f, g)$. We observe that performance is pretty close to the optimal performance even for five regions. In practice, our experience suggests using 4-6 regions should be sufficient to obtain reasonable performance.

[4]Pickle is the standard way of serializing objects in Python.

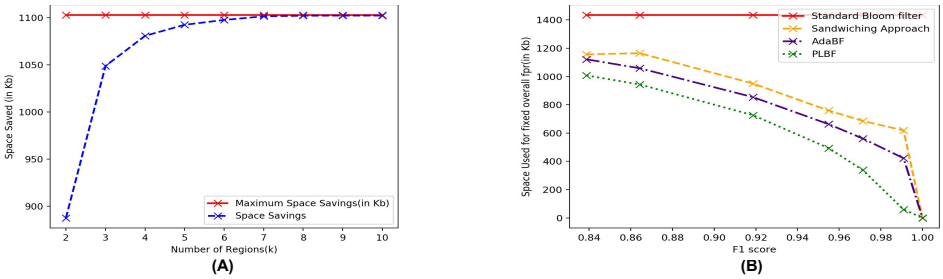


Figure 3: (A) Space saved as we increase the number of regions $k$ (B) Space occupied against F1 score for the Zipfian distribution

### 4.2.2 Performance and Model Quality

Here we provide an exemplary experiment to see how the performance of various methods vary with the quality of the model. As discussed earlier, a good model will have high skew of the distributions $f$ and $g$ towards extreme values. We therefore vary the skew parameter of the Zipfian distribution to simulate the model quality. We measure the quality of the model using the standard F1 score. Fig.3(B) represents the space required by various methods to achieve a fixed false positive rate of $0.001$ as we vary the F1 score of the model. The figure shows that as the model quality in terms of the F1 score increases, the space required by all the methods decreases (except for the standard Bloom filter, which does not use a model). The space used by all the methods goes to zero as the F1 score goes to 1, as for the synthetic dataset there is no space cost for the model. The data point corresponding to F1 score equal to 0.99 was used to plot Fig.2(A).

## 5 Conclusion

Our analysis of the partitioned learned Bloom filter provides a formal framework for improving on learned Bloom filter performance that provides substantially better performance than previous heuristics. As Bloom filters are used across thousands of applications, we hope the PLBF may find many uses where the data set is amenable to a learned model.

## References


Hyrum S. Anderson and Phil Roth. 2018. EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models. arXiv:cs.CR/1804.04637

Burton H. Bloom. 1970. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (July 1970), 422–426. https://doi.org/10.1145/362686.362692

Prosenjit Bose, Hua Guo, Evangelos Kranakis, Anil Maheshwari, Pat Morin, Jason Morrison, Michiel Smid, and Yihui Tang. 2008. On the false-positive rate of Bloom filters. *Inform. Process. Lett.* 108, 4 (2008), 210–213.

Andrei Z. Broder and Michael Mitzenmacher. 2003. Survey: Network Applications of Bloom Filters: A Survey. *Internet Math.* 1, 4 (2003), 485–509. https://doi.org/10.1080/15427951.2004.10129096

Jehoshua Bruck, Jie Gao, and Anxiao Jiang. 2006. Weighted Bloom filter. In *Proceedings 2006 IEEE International Symposium on Information Theory, ISIT 2006, The Westin Seattle, Seattle, Washington, USA, July 9-14, 2006*. IEEE, 2304–2308. https://doi.org/10.1109/ISIT.2006.261978

Zhenwei Dai and Anshumali Shrivastava. 2019. Adaptive Learned Bloom Filter (Ada-BF): Efficient Utilization of the Classifier. *arXiv preprint arXiv:1910.09131* (2019).

Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2018. Optimal Bloom Filters and Adaptive Merging for LSM-Trees. *ACM Trans. Database Syst.* 43, 4, Article 16 (Dec. 2018), 48 pages. https://doi.org/10.1145/3276980

Peter C. Dillinger and Panagiotis Manolios. 2004. Bloom Filters in Probabilistic Verification. In *Formal Methods in Computer-Aided Design, 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004, Proceedings (Lecture Notes in Computer Science)*, Alan J. Hu and Andrew K. Martin (Eds.), Vol. 3312. Springer, 367–381. https://doi.org/10.1007/978-3-540-30494-4_26

Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 489–504. https://doi.org/10.1145/3183713.3196909

Michael Mitzenmacher. 2002. Compressed bloom filters. *IEEE/ACM Trans. Netw.* 10, 5 (2002), 604–612. https://doi.org/10.1109/TNET.2002.803864

Michael Mitzenmacher. 2018. A Model for Learned Bloom Filters and Optimizing by Sandwiching. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.). Curran Associates, Inc., 464–473. http://papers.nips.cc/paper/7328-a-model-for-learned-bloom-filters-and-optimizing-by-sandwiching.pdf

Michael Mitzenmacher, Salvatore Pontarelli, and Pedro Reviriego. 2020. Adaptive Cuckoo Filters. *J. Exp. Algorithmics* 25, 1, Article 1.1 (March 2020), 20 pages. https://doi.org/10.1145/3339504

Anna Pagh, Rasmus Pagh, and S. Srinivasa Rao. 2005. An Optimal Bloom Filter Replacement. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '05)*. Society for Industrial and Applied Mathematics, USA, 823–829.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

Jack Rae, Sergey Bartunov, and Timothy Lillicrap. 2019. Meta-Learning Neural Bloom Filters. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.), Vol. 97. PMLR, Long Beach, California, USA, 5271–5280. http://proceedings.mlr.press/v97/rae19a.html

**APPENDIX**

## A Solving the Relaxed Problem using KKT conditions

As mentioned in the main text, if we relax the constraint of $fpr_i \leq 1$, using the stationary KKT conditions we can obtain the optimal $fpr_i$ values. Here we show this work. The appropriate Lagrangian equation is given in Eq.15. In this case, the KKT coniditions tell us that the optimal solution is a stationary point of the Lagrangian. Therefore, we find where the derivative of the Lagrangian with respect to $fpr_i$ is zero.

$$\begin{aligned} L\left(t_i, fpr_i, \lambda, \nu_i\right) = \sum_{i=1}^{k}\left(F(t_i) - F(t_{i-1})\right) * \log_2\left(\frac{1}{fpr_i}\right) \\ +\lambda * \left(\left(\sum_{i=1}^{k}\left(G(t_i) - G(t_{i-1})\right) * fpr_i\right) - FPR\right) \\ +\sum_{i=1}^{k} \nu_i * (t_{i-1} - t_i) \end{aligned} \tag{15}$$

$$\frac{\partial L(t_i, fpr_i, \lambda, \nu_i)}{\partial fpr_i} = 0 \tag{16}$$

$$\frac{\partial (F(t_i) - F(t_{i-1})) \log_2\left(\frac{1}{fpr_i}\right)}{\partial fpr_i} = -\lambda \frac{\partial (G(t_i) - G(t_{i-1})) * fpr_i}{\partial fpr_i} \tag{17}$$

$$fpr_i = \frac{\ln(2) * (F(t_i) - F(t_{i-1})) * \lambda}{(G(t_i) - G(t_{i-1}))} \tag{18}$$

$$\lambda = \frac{FPR}{\ln(2) * \sum_{i=1}^{k}\left(F(t_i) - F(t_{i-1})\right)} = \frac{FPR}{\ln 2} \tag{19}$$

$$fpr_i = \frac{(F(t_i) - F(t_{i-1})) * FPR}{(G(t_i) - G(t_{i-1}))} \tag{20}$$

Eq.18 expresses $fpr_i$ in terms of $\lambda$. Summing Eq.18 over all $i$ and using the relationship between $FPR$ and $G$ we get Eq.19. Thus the optimal fpr values turn out to be as given in Eq.20.

## B Algorithm for finding thresholds

We provide the pseudocode for the algorithm to find the solution for the relaxed problem; Alg.1 finds the thresholds and false positive rates. As we have described in the main text, this algorithm provides the optimal parameter values, if key and non-key densities are monotonically increasing and decreasing, respectively.

The idea is that only the false positive rate of the rightmost region can be one. The algorithm receives discretized key and non-key densities. The algorithm first iterates over all the possibilities of the rightmost region. For each iteration, it finds the thresholds that maximize the KL divergence for the rest of the array for which a dynamic programming algorithm exists. After calculating these thresholds, it finds the optimal false positive rate for each region using Alg.2. After calculating the thresholds and false positive rates, the algorithm calculates the space used by the back-up Bloom filters in PLBF. It then remembers the index for which the space used was minimal.

## C Optimal FPR for given thresholds

We provide the pseudocode for the algorithm to find the optimal false positive rates if threshold values are provided. The corresponding optimization problem is given in Eq.21. As the boundaries for the regions are already defined, one only needs to find the optimal false positive rate for the backup Bloom filter of each region.

**Algorithm 1** Solving the general problem

**Input** $F_{dis}$ - the array containing discretized key density of each region
**Input** $G_{dis}$ - the array containing discretized key density of each region
**Input** $FPR$ - target overall false positive rate
**Input** $k$ - number of regions
**Output** $t$ - the array of threshold boundaries of each region
**Output** $fpr$ - the array of false positive rate of each region
**Algorithm** ThresMaxDivDP - DP algorithm that returns the thresholds maximizing the divergence between key and non-key distribution.
**Algorithm** CalcDensity - returns the region density given thresholds of the regions
**Algorithm** OptimalFPR - returns the optimal false positive rate of the regions given thresholds
**Algorithm** SpaceUsed - returns space used by the back-up Bloom filters given threhsolds and false positive rate per region.

**procedure** SOLVE($F_{dis}, G_{dis}, FPR, k$)
  $MinSpaceUsed \leftarrow \infty$ ▷ Stores minimum space used uptil now
  $index \leftarrow -1$ ▷ Stores index corresponding to minimum space used
  $F_{last} \leftarrow 0$ ▷ Key density of the current last region
  $G_{last} \leftarrow 0$ ▷ Non-key density of the current last region

  **for** $i$ **in** $k-1, k, ...N-1$ **do** ▷ Iterate over possibilities of last region
    $F_{last} \leftarrow \sum_{j=i}^{N} F_{dis}[j]$ ▷ Calculate the key density of last region
    $G_{last} \leftarrow \sum_{j=i}^{N} G_{dis}[j]$
    $t \leftarrow$ ThresMaxDivDp($F[1..(i-1)], G[1..(i-1)], k-1$) ▷ Find the optimal thresholds for the rest of the array
    $t.append(i)$
    $F', G' \leftarrow$ CalcDensity($F_{dis}, G_{dis}, t$)
    $fpr =$ OptimalFPR($F', G', FPR, k$) ▷ Find optimal false positive rates for the current configuration
    **if** ($MinSpaceUsed <$ SpaceUsed($F_{dis}, G_{dis}, t, fpr$))
      **then** $MinSpaceUsed \leftarrow$ SpaceUsed($F_{dis}, G_{dis}, t, fpr$); $index \leftarrow i$ ▷ Remember the best performance uptil now

  $F_{last} \leftarrow \sum_{j=index}^{N} F_{dis}[j]$
  $G_{last} \leftarrow \sum_{j=index}^{N} G_{dis}[j]$
  $t \leftarrow$ ThresMaxDivDP($F[1..(index-1)], G[1..(index-1)], k-1$)
  $t.append(index)$
  $F', G' \leftarrow$ CalcDensity($F_{dis}, G_{dis}, t$)
  $fpr =$ OptimalFPR($F', G', FPR, k$)

  **return** $t$ , $fpr$ Array

$$\begin{aligned} \min_{fpr_{i=1...k}} \quad & \textstyle\sum_{i=1}^{k} (F(t_i) - F(t_{i-1})) * \log_2(\frac{1}{fpr_i}) \\ \text{constraints} \quad & \textstyle\sum_{i=1}^{k} (G(t_i) - G(t_{i-1})) * fpr_i = FPR \\ & fpr_i \leq 1 \;\; i = 1...k \end{aligned} \tag{21}$$

Alg.2 gives the pseudocode. We first assign false positive rates based on the relaxed problem but may find that $fpr_i \geq 1$ for some regions. For such regions, we can set $fpr_i = 1$, re-solve the relaxed problem with these additional constraints (that is, excluding these regions), and use the result as a solution for the general problem. Some regions might again have a false positive rate above one, so we can repeat the process. The algorithm stops when there is no new region with false positive rate greater than one. This algorithm finds the optimal false positive rates for the regions when the thresholds are fixed.

**Algorithm 2** Finding optimal fpr's given thresholds

**Input** $F'$ - the array containing key density of each region
**Input** $G'$ - the array containing non-key density of each region
**Input** $FPR$ - target overall false positive rate
**Input** $k$ - number of regions
**Output** $fpr$ - the array of false positive rate of each region

**procedure** OPTIMALFPR($F', G', FPR, k$)
  $F_{sum} \leftarrow 0$ ▷ sum of key density of regions with $fpr_i = 1$
  $G_{sum} \leftarrow 0$ ▷ sum of non-key density of regions with $fpr_i = 1$
  **for** $i$ **in** $1, 2, ...k$ **do**
    $fpr[i] \leftarrow \frac{F'[i] \cdot FPR}{G'[i]}$ ▷ Assign relaxed problem solution
  **while** some $fpr[i] > 1$ **do**
    **for** $i$ **in** $1, 2, ...k$ **do**
      **if** $(fpr[i] > 1)$ **then** $fpr[i] \leftarrow 1$ ▷ Cap the false positive rate of region to one
    $F_{sum} \leftarrow 0$
    $G_{sum} \leftarrow 0$
    **for** $i$ **in** $1, 2, ...k$ **do**
      **if** $(fpr[i] = 1)$ **then** $F_{sum} \leftarrow F_{sum} + F'[i]; G_{sum} \leftarrow G_{sum} + G'[i]$ ▷ Calculate key,non-key density in regions with no Bloom filter($fpr[i] = 1$)
    **for** $i$ **in** $1, 2, ...k$ **do**
      **if** $(fpr[i] < 1)$ **then** $fpr[i] = \frac{F'[i] \cdot (FPR - G_{sum})}{G'[i] \cdot (1 - F_{sum})}$ ▷ Modifying the $fpr_i$ of the regions to ensure target false positive rate is FPR
  **return** $fpr$ Array