A comparison of Learned Index Structures and traditional algorithms

Jens Birk Andersen - jbia@itu.dk
Malthe Mathias Mølgaard Larsen - malla@itu.dk

Course: Bachelor project

Supervisors: Martin Aumüller - maau@itu.dk,
Ioana Oriana Bercea - iobe@itu.dk

Year: Spring 2023

Course code: BIBAPRO1PE

Github repository:
https://github.com/BSc-learned-indexes/daisy-bf

Abstract

This project falls within two fundamental areas of computer science: data structures and machine learning. Learned Index Structures merges these two worlds and provides interesting new takes on old and proven data structures such as B-Trees and Hash Maps. Bloom Filters, which are used for membership testing, are no exception. Recent work has provided new approaches to optimise memory efficiency of the Bloom Filter. The new learned approaches include *The Partitioned Learned Bloom Filter* and *The Adaptive Learned Bloom Filter*. This project explores a new theoretical contender, *The Daisy Bloom Filter*. It provides the first implementation of the filter and compares it to the state-of-the-art in a transparent experimental setting. It concludes that all the Learned Bloom Filters perform better than the traditional Bloom Filter and shows that the performance of each filter vary depending on the conditions and assumptions that are in each experimental setting. It shows that the Daisy Bloom Filter does not provide a better space to false positive rate ratio compared to the other Learned Bloom Filters in the experiments throughout this project.

IT University of Copenhagen

May 15, 2023

Contents

1 Introduction

Data structures and algorithms have been studied intensively through time. Many papers have been written on fundamental data structures like range indexes, look-up tables and existence indexes.
Traditionally, data structures and algorithms do not consider the structure of the data. Instead, they are optimized for the worst-case input. Learned Index Structures are a new approach that considers the structure of the data in the form of learned model. It does so by leveraging machine learning in an attempt to optimize space and time complexity of the data structures. This project investigates learned approaches to Bloom Filters in particular. Numerous learned approaches to Bloom Filters have been proposed in the recent years. These include *A Model for Learned Bloom Filters, and Optimizing by Sandwiching* [11], *Adaptive Learned Bloom Filters* [7] and *Partitioned Learned Bloom Filters* [18]. A very recent proposal is the *Daisy Bloom Filter* [2]. Daisy Bloom Filters are theoretically superior to previous learned approaches, but lack empirical evidence to support it.
The project investigates whether Daisy Bloom Filters provide a better space complexity than the original Bloom Filter and the new learned approaches in practice. This project contributes with the very first implementation of the Daisy Bloom Filter. This implementation and it's surrounding bench-marking framework is made publicly available along with the data sets that the experiments in the project are run against. Thus, making the setup transparent and replicable for anyone interested. We benchmark the space performance of the Daisy Bloom Filter over a synthetic data set and a real world data set. At last we compare it to existing state-the-art learned approaches to Bloom Filters in the same setting.

2 The regular Bloom Filter

2.1 Introduction and properties

A Bloom Filter is a space efficient, probabilistic data structure that was introduced in 1970 by Burton H. Bloom [3]. For a given set S (the key set), the Bloom Filter is used to identify if an element x is in the set, $x \in S$ (a key), with a false positive probability F, where $0 < F < 1$. The Bloom Filter has the characteristic that it allows false positives, but not false negatives. I.e. an element $x \in S$ which is queried, will always result in a correct output, unlike $x \notin S$ (a non-key), which has a probability F of being falsely positive.
A Bloom Filter supports two types of operations: inserts and queries. The Bloom Filter is constructed by creating a bit-array with a length m. Each bit is initially 0. To insert an element x into the Bloom Filter, k independent hash functions are used to determine indexes in the array. At each index, the bits are set to 1. This indicates that the hashed member might be a part of the set S. Each hash function is assumed to be a randomized function that produces uniformly distributed hashes. The range of each hash function is $[0, m-1]$.
To determine how many hash functions are needed to insert an element into the filter, we need to look at what the acceptable false positive probability F for the filter is. This is a choice that has to take the application of the filter into consideration. The number of hash functions, given the size of the set $n = |S|$, and the length of the array m, the optimal number of hash functions (i.e. the number of hash functions that minimizes the false positive rate for a given m and n) are given by[1]:

$$k = \ln(2) \cdot \frac{m}{n} \tag{1}$$

[1]Note that going forward, we employ $\log_2$ as log and $\log_e$ as ln.

The optimal length of the bit array in the Bloom Filter m is determined by[4]:

$$m = n \cdot \log(e) \cdot \log\left(\frac{1}{F}\right) \approx n \cdot 1.44 \cdot \log\left(\frac{1}{F}\right) \tag{2}$$

Thus, the optimal number of bits per element is:

$$\frac{m}{n} = \log(e) \cdot \log\left(\frac{1}{F}\right) \approx 1.44 \cdot \log\left(\frac{1}{F}\right) \tag{3}$$

We can now yield the optimal number of hash functions by inserting (3) into (1):

$$k = \ln(2) \cdot \log(e) \cdot \log\left(\frac{1}{F}\right) = \frac{\ln(2)}{\ln(e)} \cdot \frac{\ln(e)}{\ln(2)} \cdot \log\left(\frac{1}{F}\right) = \log\left(\frac{1}{F}\right) \tag{4}$$

In order to test if an element is a member of S, the element is hashed with k hash functions, which gives k indexes to check in the bit array. If the value at any of the given indexes is 0, then we know that the element is **not** in the set. If all bits in at the indexes are 1, then the element **might** be in the set. The Bloom Filter might give a false positive due to the fact that collisions can occur during queries. This can happen if two different elements hashes to the same indexes in the array. If an element $y \notin S$ is hashed to the same indexes as an element $x \in S$, then y will be a false positive, since the bits are set to 1.
Given a single hash function with an even distribution, the probability that a bit is set to 1 will be $\frac{1}{m}$. The probability of it being a 0, will then be $1 - \frac{1}{m}$. If there is inserted n elements into the Bloom Filter with k hash functions, we can calculate the probability p_0 of a bit being a 0 by:

$$p_0 = \left(1 - \frac{1}{m}\right)^{n \cdot k} \tag{5}$$

The probability p_1 that a a given bit is set to 1 is then $1 - p_0$:

$$p_1 = 1 - \left(1 - \frac{1}{m}\right)^{n \cdot k} \tag{6}$$

At last we can derive the probability that if we check k random indexes in the array, the probability of all indexes being 1 is $(p_1)^k$. We call this the false positive probability, as this is the probability that an element $x \notin S$, will hash to k indexes, which are all 1:

$$F = \left(1 - \left(1 - \frac{1}{m}\right)^{n \cdot k}\right)^k \tag{7}$$

Further reduction of the false positive probability F can be done by inserting $\lim_{m \to \infty} \left(1 - \frac{1}{m}\right)^m = \frac{1}{e}$ into (7) yields:

$$F = \left(1 - \left(1 - \frac{1}{e}\right)^{n \cdot k}\right)^{\frac{k}{m}} \approx \left(1 - e^{\frac{-k \cdot n}{m}}\right)^k \tag{8}$$

Here we can note, that the right hand expression 8 can be minimized if the value of $k = \ln(2) \cdot \frac{m}{n}$ [4]:

$$F_{min} = \left(\frac{1}{2}\right)^k \approx 0.6186^{\frac{m}{n}} \tag{9}$$

2.2 Limitations

The Bloom Filter does not have the ability to remove an element which has been inserted. This is caused due to the fact that there would be no way to ensure that another element does not share the same bits as a result of hash collisions. If any bit shared by more than one element is removed, then the Bloom Filter cannot guarantee no false negatives. If we wish to remove an element from the set, then the Bloom Filter will have to be rebuild with the updated set.

2.3 Applications

A Bloom Filter has many applications. In general, it is a useful data structure in situations where you want to test an element for membership of a given set, and the set is large enough to make hash tables unreasonable, memory wise. Examples of this can be found in spam filters, caching, databases and many parts of networking [4]. For instance, the Google Chrome Web browser used to use Bloom Filters to store information about malicious URLs in the client browsers [6].

3 Classification

3.1 Introduction

Classification in machine learning is the concept of mapping input data to different categories (labels). In machine learning the input data is typically vectorized[2] and processed in a learned model. The model then labels the data with some degree of accuracy. The model is created by inputting training data, and then adjusting the weights of the model to improve the accuracy.
When training the model, the goal is to minimize the error or loss. This is done via. a loss function, which measure the error/loss of the predicted to the actual output.
When training the model it is normal practice to split the data set into two subsets. A training and a testing set. The training set is used to train the model, and the test set is used to evaluate the performance of the model. One reason to split the data set is to avoid overfitting the model to the training data. This is important as machine learning models can become too specific for the training data. On Figure 1 below we see an example of overfitting. This causes the model to be tuned too much to the specific training data. You can see that it classifies every dot including the outliers perfectly. However it does not capture the general "pattern" of the data.

[2]Vectorization is the act of turning various data (e.g. strings) into numbers that can be used in a ML model.

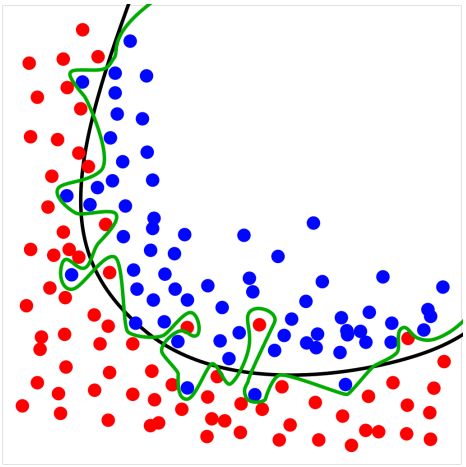


Figure 1: An example of overfitting (green line) [5].

When splitting the data set into the training and testing set, we can improve the weights of the model according to the loss. However, we stop the training if the loss of the test data begins to increase. This avoids overfitting. This is depicted in Figure 2 below.

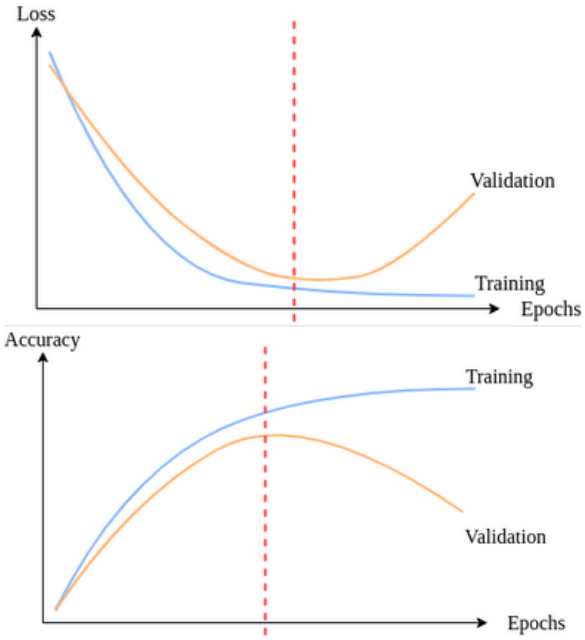


Figure 2: Loss and Accuracy as per Epochs (training cycles)[17].

Evaluation of Model Performance

There are several ways to evaluate classification models. One fundamental metric to look at is the Confusion Matrix. It consists of four fields: True Positives (TP), False Positives (FP), True Negatives (TN) and False Negatives (FN).

		Actual	
		Positive	Negative
Prediction	Positive	TP	FP
	Negative	FN	TN

Table 1: Confusion Matrix.

The total number of positives, which where predicted to be positive by the model are called True Positives (TP). The sum of the elements that the model predicted to be positive but were actually negative are the False Positives (FP). The total number of negatives the where correctly predicted to be negatives are the True Negatives (TN). At last, the total of the elements that were predicted to be negative but actually were positive are the False Negative (FN).
These four terms are useful for constructing more performance indicators. The first is the model's *accuracy*. Accuracy is calculated by dividing the *number of correct predictions* with the *total number of predictions*:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{10}$$

Other methods of evaluation is *precision* and *recall*. Precision is a metric of what fraction of the positive predictions where actual positives. This is a useful metric when evaluating models on the ratio of true positives to total positives. Precision is especially useful for models where we want to reduce the false positives, like the Bloom Filter, as a high precision is a sign of a low ratio of false positives relative to true positives.

$$Precision = \frac{TP}{TP + FP} \tag{11}$$

Recall is a metric of what fraction of the actual positives, where predicted as positives:

$$Recall = \frac{TP}{TP + FN} \tag{12}$$

Recall is a useful metric if the model is to be evaluated on how well it can label actual positives. This is important for models where false negatives can have consequences. For a model detecting tumors it is more important to have a high recall, than it is to have a high precision, as it is better to detect the tumors, than to mislabel a non-tumor as a tumor.
If we want to evaluate the model based on both Precision and Recall we can use the F1-score, which is the harmonic mean of the Precision and Recall:

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{13}$$

We use these metrics to evaluate the performance of the classification models that are used in the learned Bloom Filters.

3.2 Logistic Regression

Logistic regression is a commonly used model for binary classification. Despite it's deceiving name, it's actually a classification model and not a regression model. It can be used both for binary classification and class probability estimation. The logistic regression model takes a linear combination of features[3]. These are passed to a non-linear sigmoidal[4] function [1]. The sigmoial function is given by: $S(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1}$.

3.3 Random Forest Classifier

Another classification model, which is used throughout the experiments in this project, is the Random Forest Classifier. The Random Forest Classifier combines Decision Trees to avoid overfitting.

Decision Trees

Decision Trees are binary trees where each node, split the data set into two partitions. It does so based on a threshold-value for a specific feature within the data set. Combining multiple partitioning nodes can then be used to recursively split the data set into partitions. The partitions can then be labeled, making it possible to classify an element. To train the Decision Tree model, each split is chosen to improve the loss function. For the implementation which Scikit-learn employs, the loss function is measuring the impurity of the splits[12].

Random Forest Classifier

The Random Forest Classifier[14] is based on Decision Trees. To reduce overfitting and improve predictive accuracy it consists of multiple Decision Trees. The trees are trained on a randomly chosen subset of the training data, hence the name "Random Forest". When classifying an element, each Decision Tree in the forest outputs a label for the element. The label with the majority is then chosen. This generalizes the classification better than a Decision Tree, as each Decision Tree in the Random Forest Classifier is not exposed to the full data set. This makes it difficult to overfit the training data.
The Random Forest Classifier can be tuned to increase accuracy with different hyperparameters. In the experiments we have tuned the parameters: *n_estimators* and *max_leaf_nodes* [14]. *n_estimators* change the number of trees in the model. This results in a better accuracy as there are more Decision Trees to capture the variation of the data. Increasing or decreasing this variable will change the space usage of the model by a factor of M in the space usage of the model:

$$\mathcal{O}(M \cdot N \cdot \log(N)) \tag{14}$$

(14) shows space usage by the default implementation of the Random Forest Classifier from Scikit-learn [13]. The M is the number of trees in the model and the N is the number of samples.
The parameter, *max_leaf_nodes*, controls the maximum number of leaf nodes per Decision tree. This means that the tree cannot grow past a certain amount of nodes. Thus, limiting the amount of space and accuracy, as more leaf-nodes should yield better accuracy at the cost of memory.
To get the predicted probability of an element being labeled as x, the average of the output score of each Decision Tree is used.

[3] A linear combination of features combines a series of features $x_i, x_{i+1}...x_n$ that each have an associated weight $w_i, w_{i+1}...w_n$. Thus, the linear combination y of these features is: $y = x_i \cdot w_i + x_{i+1} \cdot w_{i+1} + ... + x_n \cdot w_n$

[4] A sigmoial function is an S shaped function. It takes an input x and maps it to an output in the range [0,1]. The output can be interpreted as a probability.

4 Learned Index Structures

In 2018 the paper *The Case for Learned Index Structures*[9] introduced a new approach to building data structures and algorithms. The new approach is to optimize data structures and algorithms with the use of machine learning techniques. The idea is to find patterns / features in the specific data set and use these in the optimization of specific algorithms and data structures.
The paper suggests that opportunities for optimizations are found in a lot of different fundamental areas such as range indexes (B-trees), point indexes (Hash Maps) and existence indexes (Bloom Filters).

4.1 Example of an optimization

To illustrate the rationale behind the idea, the authors of [9] explore the possibility of improving the B-tree for a very specific data set. The specific data set for this example consists of a series of time stamps. We know from the nature of the data, that $x_{i-1} < x_i, \forall i$. Then we might be able to use the timestamp as a index, and the B-tree is reduced to a lookup table. Thus, we can achieve a lookup complexity of $\mathcal{O}(1)$ instead of $\mathcal{O}(\log n)$.
Even though this example is very idealized and simple it illustrates the point: given information about the data, that a given instance of a data structure is used upon, opportunities of improvements are now possible with the advancements of machine learning techniques.

4.2 Limitations and challenges

There are various different limitations for Learned Index Structures. Some Learned Index Structures like the B-tree replacement do not support inserts without having to retrain the model.

Since machine learning techniques are fundamental for the performance of the learned indexes, the model's performance determines how well the learned index perform. For instance, a Learned Bloom Filter with an architecture where a ML model acts as a pre-filter can only achieve as good as a false positive rate as the ML model is able to produce.

In addition to this, there is also a trade-off between the ML model's performance and the it's size. As an example, the size of a Random Forest Classifier can be dramatically reduced if the height of the forest is limited. However, this results in a worse classification. How these trade-offs are to be balanced is not solved or discussed in the mentioned papers.

A lot of traditional data structures have also been cache optimized. Various other tricks like utilizing `SIMD` registers are much easier to utilize in a traditional data structure than in ML models. For instance, a neural network require all weights to make a prediction. In contrast, B-trees keep the top nodes (which are used all the time) in the cache [9].

4.3 Bloom Filter as a classification problem

The main function the Bloom Filter has is to check if an element is a member of a set. When an element is checked for membership on a Bloom Filter it can result in two outcomes. Either it is **not** a member of the set, or it **might** be a member of the set, i.e. no false negatives, only false positives. This can be loosely translated to a binary classification problem. Here a classification model can, with some error, label an input element as a member or a non-member. One main distinction of the classification compared to the Bloom Filter, is that the classification can introduce false positives

and **false negatives**. The introduction of false negatives is not a desired attribute of the Bloom Filter. This is why in the following sections the Learned Bloom Filters use an underlying Bloom Filter to check if any **labeled** non-members (possible false negatives), are **true** non-members (true negatives).

5 The Original Learned Bloom Filter

The original Learned Bloom Filter was proposed in 2018 and serves as an alternative way to think about Bloom Filters. It uses a machine learning model as pre-filter to the underlying Bloom Filter to reduce the space of the bit array.

5.1 Design

The design of the Learned Bloom Filter is simple. It consists of a machine learning model and a traditional Bloom Filter which serves as a backup filter. The machine learning model gives each element a score s_x. The score is the predicted probability, that the element is in the set S.
A threshold τ is chosen. All elements with a model score that is above τ, the filter says `Yes`. This will result in false positives for elements which are not included in the set, and which have a score above the threshold. To have the same properties as a regular Bloom Filter, The Learned Bloom Filter must not produce any false negatives. For $s_x > \tau$ the filter only says `Yes`, this doesn't produce any false negatives, meaning the possibility of a false negative doesn't arise. For $s_x < \tau$ all the keys are stored in a regular Bloom Filter. The regular Bloom Filter doesn't produce any false negatives. If the model says `No`, then the backup filter is asked. If the backup Bloom Filter says `Yes` then we answer `Yes` (still with the possibility of a false positive). If the backup Bloom Filter says `No`, then we're certain that the element that is tested for membership is not part of the set and there are no false negatives.

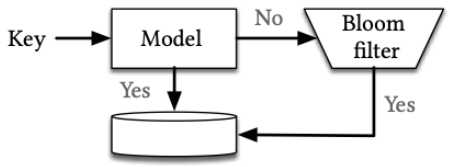


Figure 3: Original Learned Bloom Filter [9].

The size of the backup Bloom Filter is dependant on the value of $\tau \in [0, 1]$. If $\tau = 1$ then the amount of space allocated for the bit array will be the same as for the regular Bloom Filter. Choosing a high τ will lead to a lower false positive probability, but at the cost of more space allocated for the bit array. On the other hand, choosing a low τ will increase the amount of false positives that comes from the model.
One interesting property of the Learned Bloom Filter is that the size of the model is independent of the size of the set. The size of the backup filter proportional with the size of the false negative rate of the machine learning model. Whereas the standard Bloom Filter size scales with the set size, as discussed in section 2. This means that inserting more keys which have a score above the threshold, will increase the regular Bloom Filter's size, but the Learned Bloom Filter will remain the same

size. If this is the case, then the Learned Bloom Filter has an advantage compared to the standard Bloom Filter.

6 Adaptive Learned Bloom Filters

An improvement on the Original Learned Bloom Filter is described in the paper: *Adaptive Learned Bloom Filter (Ada-BF): Efficient Utilization of the Classifier*[7].
The improvement over the Learned Bloom Filter comes from the notion that the density of keys and non-keys at a given score, is inverse proportional. This notion can be used to introduce more thresholds, which partition the scores into several groups. The authors propose two different ways of constructing Learned Bloom Filters where the scores are utilized better than the original Learned Bloom Filter.

6.1 Ada-BF

The first approach is called Ada-BF. For the Ada-BF the partitions of the scores determine how many hash functions each element should be associated with. The number of hash functions are then used to insert and/or query elements. Figure 4 shows how the elements are partitioned by their score into groups G, from G_1 to G_g. The elements belongs to a group G_i if the scores for the elements, are within the thresholds, τ, for the group: $x \in G_i$ if $\tau_{i-1} \leq score(x) < \tau_i$. For each group the elements have a number of hash functions which determine how many bits should be checked or set for queries and inserts.

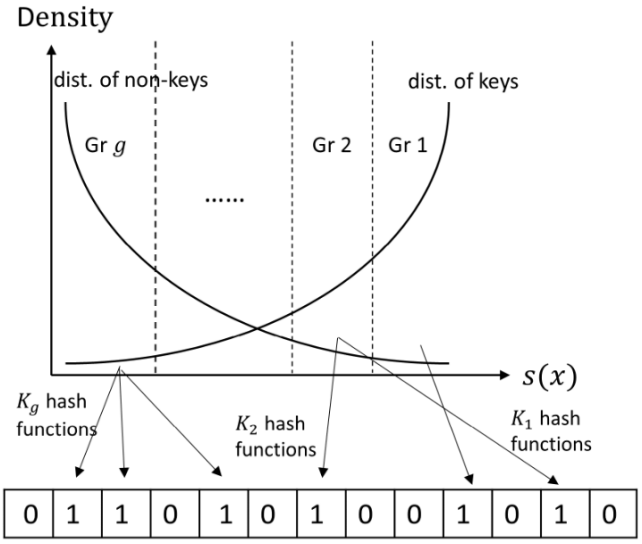


Figure 4: Adaptive Learned Bloom Filter [7].

From (6) we know that the probability of a bit being set to 1 in a regular Bloom Filter depends of number of hash functions, k, number of keys, n, and size of the bit array, m. This can be expressed for the Ada-BF as the following expression:

$$p_1 = 1 - \left(1 - \frac{1}{m}\right)^{\sum_{t=1}^{g} n_t \cdot k_t} \tag{15}$$

where $n_t$is the number of keys in G_t
k_t is the number of hash functions for G_t

This gives us the probability of a bit being set to 1 in the Ada-BF bit array.
To find the false positive probability f_i for a group, G_i, which uses k_i hash functions, we must check k_i bits in the array, this gives us:

$$f_i = \left(1 - \left(1 - \frac{1}{m}\right)^{\sum_{t=1}^{g} n_t \cdot k_t}\right)^{k_i} = (p_1)^{k_i} \tag{16}$$

The false positive probability for the whole Ada-BF can be expressed as the sum of all the groups false positive probability, multiplied with the probability that a non-key is from that group:

$$F = \sum_{t=1}^{g} Pr_i \cdot f_i = \sum_{t=1}^{g} Pr_i \cdot (p_1)^{k_i} \tag{17}$$

where Pr_i is the probability that a non-key is in the group G_i

Parameter Tuning

To build the Ada-BF we need to know the values of the thresholds and number of hash functions for each group. To minimize the false positive rate in reasonable time, the authors suggest keeping the estimated false positive rate fixed across all groups. They also make the observation that an increase k_i in (16) reduces the false positive rate exponentially. Because of the observation that the number of non-keys for a given score increases with the decrease in score, we can achieve similar false positive rates across all groups if we grow Pr_i exponentially with k_i, as scores decreases.
To estimate Pr_i we can do the following: count all non-keys $x \notin S$, within the group G_i. We denote this count as H_i. We then divide by the total amount of non-keys and get h_i, which estimates Pr_i:

$$h_i = \frac{H_i}{\sum_{j=1}^{g} H_j} \tag{18}$$

To simplify the parameter tuning we can fix a non-key ratio: $\frac{h_i}{h_{i+1}} = c$ and the hash functions: $k_i - k_{i+1} = 1$ for $i = 1, 2, ..., g-1$. This enables us to increase h_i exponentially with k_i. Furthermore we set $k_{min} = k_g = 0$.
Given that $\frac{Pr_i}{Pr_{i+1}} = c_i \geq c > 1$ and $k_i - k_{i+1} = 1$, we can calculate the expected false positive rate as:

$$\mathbb{E}(\text{FPR}) = \sum_{i=1}^{g} Pr_i \cdot (p_1)^{k_i} \approx \frac{\sum_{i=1}^{g} c^{g-i} \cdot (p_1)^{k_i}}{\sum_{i=1}^{g} c^{g-i}} \tag{19}$$

6.2 Disjoint Ada-BF

The second approach the authors present, is using the scores to group the elements into disjoint or separate Bloom Filters. This means that for a group G_i the elements within that group would be queried or inserted in a Bloom Filter specific to G_i. This is visualized in Figure 5

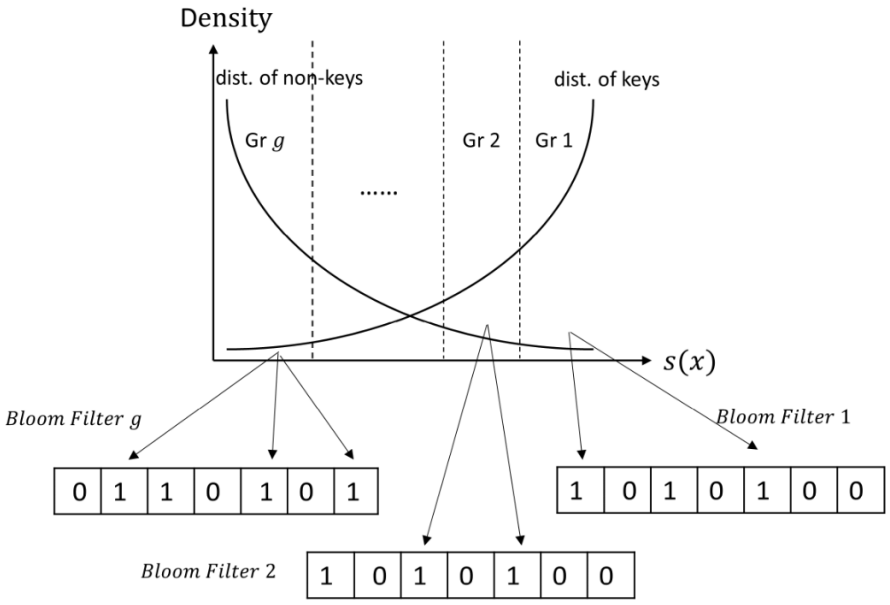


Figure 5: Disjoint Adaptive Learned Bloom Filter [7].

For the disjoint Ada-BF we know how many keys n are within each group. If we also know the space m, allocated for the Bloom Filter for the group then we can calculate the optimal number of hash functions for the group, as for the standard Bloom Filter in (1):

$$k_i = \frac{m_i}{n_i} \cdot \ln(2) \tag{20}$$

Knowing the number of hash functions k for the group, we can calculate the false positive probability for the group, like the regular Bloom Filter in equation 7:

$$f_i = \left(1 - \left(1 - \frac{1}{m_i}\right)^{n_i \cdot k_i}\right)^{k_i} \tag{21}$$

To get the false positive probability for the disjoint Ada-BF we can then sum the probability that an element is from a group G_i multiplied with the false positive probability f_i for the group, as in (17):

$$F = \sum_{t=1}^{g} Pr_i \cdot f_i \tag{22}$$

7 Partitioned Learned Bloom Filter

Partitioned Learned Bloom Filter (PLBF) is an alternative approach to building Bloom Filters that was proposed in 2020 [18]. It's approach is more granular than the Original Learned Bloom Filter approach as it's having a series of different backup filters for different thresholds.

7.1 Design

The Partitioned Learned Bloom Filter is divided into k different regions that span the scores from 0 to 1. Each region has it's associated threshold value τ_k and a target false positive rate f_k.

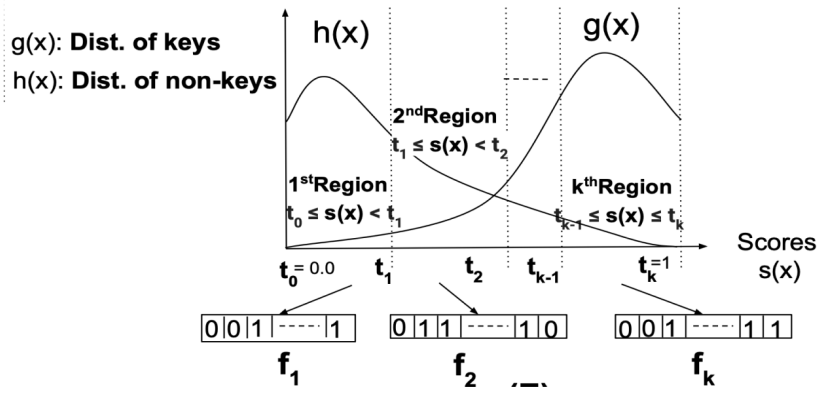


Figure 6: Partitioned Learned Bloom Filter with k different regions split up by τ_k ($\tau_k = t_k$)[18].

For the analysis, [18] denotes the distribution of keys as $g(x)$ and the non-keys as $h(x)$ as shown in Figure 6. Furthermore, $G(x)$ will denote the amount of keys that are under the curve, $G(x) = \int_0^1 g(x)\,dx$. This can be seen as the cumulative distribution function (CDF), whereas $g(x)$ can be seen as the density function. At last we let $G(\tau), H(\tau)$ denote the amount of keys and non-keys that have scores below τ. All the functions are seen as continuous in the first part of the analysis.

7.2 A constrained optimization problem

Two things need to be answered at this point: for each region, where is the optimal threshold value τ_k and what is the optimal false positive rate target for the given region.
This problem is stated as the following optimization problem:

$$\min_{\tau_i, f_i} \left(\sum_{i=1}^{k} n \cdot (G(\tau_i) - G(\tau_{i-1}) \cdot c \cdot \log\left(\frac{1}{f_i}\right) \right) \tag{23}$$

Constraint 1: $\sum_{i=1}^{k} (H(\tau_i) - H(\tau_{i-1}) \cdot f_i \leq F$
Constraint 2: $0 \leq f_i \leq 1, i = 1..k$
Constraint 3: $\tau_i - \tau_{i-1} \geq 0, i = i..k; \tau_0 = 0; \tau_k = 1$

The constraints mean that, first the sum of the false positive rate for all the backup Bloom Filters must be below the target false positive rate F. Second, that the false positive rate for each backup filter f_i is less than or equal to 1. And at last that each region is increasing (e.g. moving to the right on Figure 6) and that the first region is at 0 and the last at 1 - meaning that they cover the entire span of the range of the ML model.
Essentially, we want to minimize the space of the sum of all backup Bloom Filters. The c denotes a constant that is different for various Bloom Filter types ($\log(e) \approx 1.44$ for a regular Bloom Filter).

7.3 Solution to the optimization problem

The optimal false positive rate for a given region f_i is derived using the Karush–Kuhn–Tucker (KKT) conditions[5]. The result is the following:

$$f_i = F \cdot \frac{G(\tau_i) - G(\tau_{i-1})}{H(\tau_i) - H(\tau_{i-1})} \tag{24}$$

That is, the target false positive rate for the filter as a whole F multiplied with the fraction of keys in a given region divided by the probability that a given non-key is in the region. You can view this as how many keys to non-keys there are in a given region.

We can now replace f_i in (23) with (24), and we are left with an expression that is depending on τ_i[6]:

$$\begin{aligned} &\min_{\tau_i, f_i} \left(\sum_{i=1}^{k} n \cdot (G(\tau_i) - G(\tau_{i-1})) \cdot c \cdot \log_2 \left(\frac{1}{F \cdot \frac{G(t_i) - G(t_{i-1})}{H(t_i) - H(t_{i-1})}} \right) \right) \\ &= \\ &\min_{\tau_i, f_i} \left(\sum_{i=1}^{k} n \cdot (G(\tau_i) - G(\tau_{i-1})) \cdot c \cdot \log_2 \left(\frac{H(\tau_i) - H(\tau_{i-1})}{F \cdot (G(\tau_i) - G(\tau_{i-1}))} \right) \right) \end{aligned} \tag{25}$$

At this point we'll refer to the two distributions G and H as discrete probability distributions. The Standard KL divergence between the two distributions G and H are:

$$D_{KL}(G||H) = \sum_i G(i) \log \frac{G(i)}{H(i)} = -\sum_i G(i) \log \frac{H(i)}{G(i)} \tag{26}$$

where
$G(i) = G(\tau_i) - G(\tau_{i-1}))$
$H(i) = H(\tau_i) - H(\tau_{i-1})$

[5]The KKT conditions are an extension of the Lagrange multiplier method for constrained optimization problems.
[6]Note that we still need to add the size of the ML model in (25), yet this is not something we can optimize in terms of finding the correct parameters for τ_k and the optimal false positive rate target for each region. Thus, it's not included here.

The KL divergence is a measure of the relative entropy between the two distributions e.g. the "distance" between the key and non-key distributions. Note that this is an a asymmetric measure, that is $D_{KL}(G||H) \neq D_{KL}(H||G)$.

Finally, we can calculate the space of all the backup filters:

$$c \cdot \left(n \cdot \log\left(\frac{1}{F}\right) - n \cdot D_{KL}(G(i), H(i)) \right) \tag{27}$$

7.4 The solution in practice

We have now arrived at a place where we i) have a formula for finding the optimal threshold values t_k (27) and ii) have a way of finding the optimal false positive rate target for a given threshold (23). The algorithm is defined recursively[7]:

$$DP_{KL}(n, j) = max\left(DP_{KL}(n - i, j - 1) + \left(\sum_{r=1}^{n} g'(r) \cdot \log\left(\frac{\sum_{r=i}^{n} g'(r)}{\sum_{r=i}^{n} h'(r)} \right) \right) \right) \tag{28}$$

Where $g'(r)$ denotes the number of keys within the region r. $h'(r)$ denotes the number of non-keys within the region r.

8 Daisy Bloom Filters

The Daisy Bloom Filter is theoretical paper that, according to the authors, is an improvement upon the previously mentioned Learned Bloom Filters[2]. Its advantage over the other Learned Bloom Filters is that is utilizes a query distribution as well as the key distribution. The reason that the query distribution has an impact on the performance of the Bloom Filter, is that the false positive rate is linked to the query distribution.

8.1 Design

The Daisy Bloom Filter introduces two distributions: $\mathcal{P}$ and $\mathcal{Q}$ over the universe $\mathcal{U}$. Each element in the universe is then associated with a score p_x and q_x, which denotes the probability of drawing $x \in \mathcal{U}$ from $\mathcal{P}$ and $\mathcal{Q}$. $\mathcal{P}$ is the probability distribution of the elements being a key. $\mathcal{Q}$ is the query distribution of the elements. Hence p_x is associated with the probability of an element x being a key, and q_x is associated with the probability of x being queried. Furthermore, a set $\mathcal{P}_n$, which denotes n independent draws from $\mathcal{P}$, is introduced as the key set $S \subseteq \mathcal{U}$.
The scores p_x and q_x are assumed to be accessible from a model, which can retrieve p_x and q_x in constant time. This could be a classification model, as the other learned Bloom Filters have used.
The scores p_x and q_x are used to calculate the number of hash functions to be associated with each element x, whenever that element is inserted or queried. This makes it possible to vary the number of hash functions to increase the performance in respects to the size and false positive rate.

[7] In 7.3 we have made one important assumption that is not realistic. Namely, that the false positive rate for a given region can be above 1. In practice, whenever a regions optimal false positive rate exceeds 1, it is reset to in the algorithm and the optimal false positives rates for all other regions are recalculated.

8.2 Guarantees

From Theorem 1 in the Daisy paper[2], the authors describe that the Daisy Bloom Filter has a high probability that the false positive rate is at most F, when $\mathcal{P}$ and $\mathcal{Q}$ satisfy:

$$\sum_{x \in \mathcal{U}} p_x \cdot q_x \leq \frac{F}{n} \tag{29}$$

Because we the maximum number of hash functions associated with each element is at most $\log(\frac{1}{F})$[2], the worst case time complexity of an insert or query is at most $\mathcal{O}(\log(\frac{1}{F}))$.
Lastly the space requirements to achieve a false positive rate F is at most $\log(e) \cdot LB(\mathcal{P}_n, \mathcal{Q}, F)$ bits.

8.3 Partitioning of the Universe

The authors take advantage of the fact that each element x has an associated p_x and q_x. This is used to calculate the number of hash functions k_x used to insert or query the element. The set of the universe $\mathcal{U}$ is then partitioned into 5 partitions based on the p_x and q_x scores. Each partition $\mathcal{U}_i$, with an element $x \in \mathcal{U}_i$ has a function to calculate the k_x associated with x. The partitions are defined as:

$$\begin{aligned}
\mathcal{U}_0 &\triangleq \left\{ x \in \mathcal{U} \middle| q_x \leq F \cdot p_x \right\}, \\
\mathcal{U}_1 &\triangleq \left\{ x \in \mathcal{U} \middle| q_x > F \cdot p_x \text{ and } p_x > \frac{1}{n} \right\}, \\
\mathcal{U}_2 &\triangleq \left\{ x \in \mathcal{U} \middle| F \cdot p_x < q_x \leq min\left(p_x, \frac{F}{n}\right) \right\}, \\
\mathcal{U}_3 &\triangleq \left\{ x \in \mathcal{U} \middle| q_x > p_x \text{ and } \frac{F}{n} \geq p_x \right\}, \\
\mathcal{U}_4 &\triangleq \left\{ x \in \mathcal{U} \middle| q_x > \frac{F}{n} \text{ and } \frac{F}{n} < p_x \leq \frac{1}{n} \right\}
\end{aligned} \tag{30}$$

Elements within each universe are assigned k_x differently:

$$k_x \triangleq \begin{cases} 0 & \text{if } x \in \mathcal{U}_0 \cup \mathcal{U}_1, \\ \log(\frac{q_x}{F \cdot p_x}) & \text{if } x \in \mathcal{U}_2, \\ \log(\frac{1}{F}) & \text{if } x \in \mathcal{U}_3, \\ \log(\frac{1}{n \cdot p_x}) & \text{if } x \in \mathcal{U}_4 \end{cases} \tag{31}$$

The two definitions (30) and (31) are visualised in Figure 7 below:

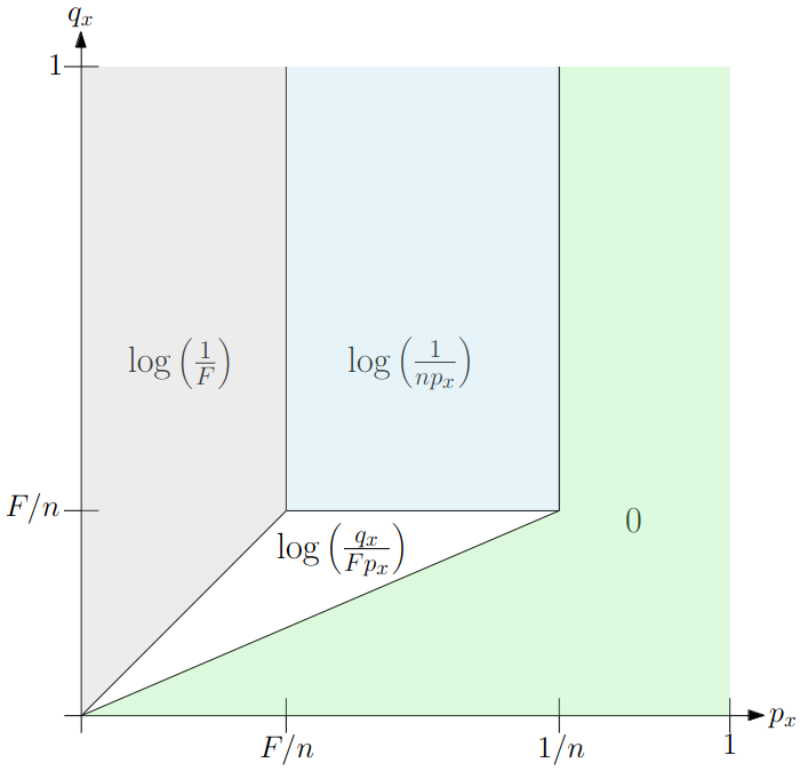


Figure 7: A visualization of the partitioned universe, with the associated k_x[2].

Elements $x \in \mathcal{U}_0$ are queried less than the probability of x being a key multiplied with the false positive probability. This means that elements within $\mathcal{U}_0$ are either queried rarely, or have a high probability of being a key.

For $x \in \mathcal{U}_1$ each x has a high probability of being a key and is queried more than the probability of it being a key, multiplied with the false positive probability. This means that $\mathcal{U}_1$ contains x with a high probability being part of the key set.

Due to the properties of $x \in \mathcal{U}_0 \cup \mathcal{U}_1$ we allow these to have 0 hash functions associated with them. This can be seen as the green area in Figure 7. We justify this, since they do not contribute much to the false positive rate of the Daisy Bloom Filter.
For keys in these universes, we do not allow them to set bits, as we rely on the model to classify them correctly.
For non-keys they are queried very rarely, we can tolerate the false positive rate introduced by these. This means that elements are not included in the filter on inserts and that we're relying on the model being able to classify them correctly.

Elements $x \in \mathcal{U}_2$ end up in the white area in Figure 7. Here the probability of querying an element has to be below $\frac{F}{n}$ and the probability that an element is a key between 0 and $\frac{1}{n}$. This can be seen as an intermediate area between the green and the gray area. This area captures elements that are

not queried often and have a probability of being a keys, which is less than elements in $\mathcal{U}_0 \cup \mathcal{U}_1$.

Elements $x \in \mathcal{U}_3$ are queried more often than the probability of it being a key. In addition to this, the probability of it being a key is less than $\frac{F}{n}$. This means that it is unlikely to be a key. However, even though it is unlikely to be a key, if we increase queries of non-keys, we expect to see an increase in the false positive rate.
To counter this we assign more hash functions for these elements. We can do this, because we expect a small amount of keys to be within this area. This means that we do not over saturate the bit array with 1's. This is what can be seen as the gray area in Figure 7. Where we give each element $k_x = \log(\frac{1}{F})$ hash functions. This is the same as the optimal number of hash functions for a regular Bloom Filter.

Elements $x \in \mathcal{U}_4$ end up in the blue area in Figure 7. An element ends up here if it has a probability of being queried more than $\frac{F}{n}$ which is quite often. At the same time it has a probability of being a key within $\frac{F}{n}$ and $\frac{1}{n}$. You can view this as being for elements that fall in between the maximum and minimum number of hash functions, and assigns hash functions according to the probability of the element being a key.

8.4 Implementation

The authors of Daisy Bloom Filters [2] do not specify an algorithm for implementing a Daisy Bloom Filter explicitly. However, they do describe the Daisy Bloom Filter in enough detail for us to be able to translate it into python-like syntax as a basis to build an implementation from[8].
There are also no details on how to access p_x or q_x as this is specific to the environment of which the Daisy Bloom Filter is implemented, but we assume there is access to these.

8.4.1 Size Allocation

To create the Daisy Bloom Filter, we must know the size of the bit array to allocate. This can be calculated from the equation:

$$DB(\mathcal{P}_n, \mathcal{Q}, F) \triangleq \log(e) \cdot LB(\mathcal{P}_n, \mathcal{Q}, \frac{F}{6}) \tag{32}$$

$LB(\mathcal{P}_n, \mathcal{Q}, F)$[9] is defined by the equation:

$$LB(\mathcal{P}_n, \mathcal{Q}, F) = n \cdot \left(\sum_{x \in \mathcal{U}_2} p_x \cdot \log\left(\frac{q_x}{F \cdot p_x}\right) + \sum_{x \in \mathcal{U}_3} p_x \cdot \log\left(\frac{1}{F}\right) + \sum_{x \in \mathcal{U}_4} p_x \cdot \log\left(\frac{1}{n \cdot p_x}\right) \right) \tag{33}$$

To simplify the implementation (33) can be simplified to:

$$LB(\mathcal{P}_n, \mathcal{Q}, F) = n \cdot \sum_{x \in \mathcal{U}} p_x \cdot k_x \tag{34}$$

Where k_x is defined as in (31).

[8]Note that in this section we expect that the reader knows how a regular Bloom Filter is implemented. Thus, this section will not describe how to create, or access bits in the underlying bit array.

[9]The parameter F is different from the target false positive rate for the Daisy Filter, as (33) guarantees a false positive rate of at most $6 \cdot F$[2].

This equation takes the sum of the probability that we choose an element x to be in the key set $\mathcal{P}_n$ multiplied with how many hash functions, k_x, that element is going to employ to set bits. This is the average number of hash functions each key employs. This is then multiplied with the number of independent draws, n, from the set $\mathcal{P}$, which corresponds to $|\mathcal{P}_n|$. Thus we multiply the average number of hash functions associated with a key, with the number of keys we insert.

```
def get_kx(x, px, qx, n, F):
    if qx <= F * px or px > 1/n: # u0 & u1
        kx = 0
    elif F * px < qx and qx <= min(px, F/n): # u2
        kx = log2(qx/(F*px))
    elif qx > px and F/n >= px: # u3
        kx = log2(1/F)
    elif qx > F/n and (F/n < px and px <= 1/n): # u4
        kx = log2(1/(n*px))
    return ceil(kx)

def LB(U, n, P, Q, F):
    sum = 0
    for x in U:
        px = P.get_px(x)
        qx = Q.get_qx(x)
        sum += px * get_kx(x, px, qx, n, F)
    return n * sum

def daisy_size(U, n, P, Q, F):
    return ceil(log2(e) * LB(U, n, P, Q, F/6))
```

Listing 1: Daisy Bloom Filter functions for calculating the size of the bit array

8.4.2 Construction

Once we know the size of the bit array, we can allocate space for the bit array, and then precede to insert the keys into the array using k_x hash functions.

```
def insert(self, x, kx):
    for ki in range(kx):
        index = get_hash(ki, x) % self.size
        self.bit_array.set_bit(index)

def __init__(U, Pn, P, Q, F):
    self.P = P
    self.Q = Q
    self.F = F
    self.n = len(Pn)
    self.size = daisy_size(U, self.n, P, Q, F)
    self.bit_array = allocate_bits(self.size)

    for key in Pn:
        px = P.get_px(key)
```

```
        qx = Q.get_qx(key)
        kx = get_kx(key, px, qx, self.n, F/6)
        self.insert(key, kx)
```

Listing 2: Functions to create a Daisy Bloom Filter.

8.4.3 Queries

To query an element we must use k_x hash functions, to check k_x bits in the bit array.

```
def query(self, x):
    px = self.P.get_px(x)
    qx = self.Q.get_qx(x)
    kx = get_kx(x, px, qx, self.n, self.F/6)
    for i in range(kx):
        index = get_hash(i, x) % self.size
        if self.bit_array.get_bit(index) == 0:
            return False
    return True
```

Listing 3: Function to query a Daisy Bloom Filter.

8.5 Theoretical advantages and disadvantages of the Daisy Bloom Filter

A theoretical advantage the Daisy Bloom Filter has over the others is the additional information gained from the query distribution. This should enable the Daisy Bloom Filter to outperform the other Bloom Filters, in regards to minimizing memory consumption at at a given false positive rate. Another advantage is that the Daisy Bloom Filter does not have to recalculate new thresholds whenever the key distribution changes. This is a great improvement over the Partitioned Learned Bloom Filter and Adaptive Learned Bloom Filter, as recalculating the thresholds is computationally hard. A disadvantage of the Daisy Bloom Filter is that the Daisy Bloom Filter requires access to the query distribution. This might limit the settings of which the Daisy Bloom Filter can be applied as this information might be lacking, or gaining access to such information might not come free of cost in terms of storage or computing. This additional cost could reduce the performance of the Daisy in comparison to others which do not have the additional cost.

9 Data sets

9.1 Malicious URLs data set

9.1.1 Source and structure of the data

The first benchmarking data set consists of Malicious URLs from Google's Transparency Report [10]. The data set contains URLs that have been labeled either benign or malicious as well as a label (0 or 1) that indicates if it was actually benign or malicious.
The data set contains 450,176 unique URLs in total. 77% of the URLs has been labeled as benign and 23% as malicious. The result column reveals that 345,738 of the URLs are actually benign and the remaining 104,438 are malicious.[10]

[10]This data set has specifically been chosen to replicate the experiments in [7] and [18]. Note that the data sets are not completely identical.

Malicious URLs Data set			
Index	URL	Label	Result
0	https://www.google.com/	benign	0
1	http://webmasteradmin.ukit.me/	malicious	1
...	...	...	...

Figure 8: Structure of Malicious URLs Data set.

9.1.2 Feature extraction

The data set consists of a URL that is represented as a string. machine learning models operate in numbers, this means that a meaningful conversation has to happen. To achieve this, a technique called *vectorization* is utilized.[11]
The URLs have been divided into 17 different Lexical Features for the classification.

17 Lexical Features		
No.	Feature	Type
0	URL	Length
1	Hostname	Length
2	Path	Length
3	First Dir.	Length
4	Top Level Dir.	Length
5	-	Count
6	@	Count
7	?	Count
8	%	Count
9	.	Count
10	=	Count
11	http	Count
13	https	Count
14	www	Count
15	Has IP address	Binary
16	Is a short URL	Binary

Figure 9: Overview of Lexical features.

9.2 Synthetic data set

In addition to using a real world data set, a synthetic data set has been created. The synthetic data set is created by drawing n times from a Zipfian distribution.

9.2.1 Zipfian distribution

Zipf's law origins from the study of natural language. It states that in many languages, the frequency of a word can be described mathematically as $f(r) \propto \frac{1}{r^{\alpha}}$, meaning that the frequency of a given word

[11] The vectorization technique has been heavily inspired by [10] in an attempt to have comparable data set as the basis to train to machine learning models on [7] and [18] that both do not fully disclose how their data sets and how their ML models are built.

which has a rank r in a frequency table is inversely proportional to it's rank[19] and some constant α. As discussed in [18] you can simulate a less accurate model by increasing the skew parameter. For our purpose we have used $\alpha = 1.0001$.

9.2.2 Producing non-keys

We draw n samples from a Zipf distribution from the `numpy.random.zipf` library. We transform each draw to have a range within 0 to 1 by dividing by the maximum value of the draws.

9.2.3 Producing keys

We draw n samples from a Zipf distribution from the `numpy.random.zipf` library. We transform each draw to have a range within 0 to 1 by dividing by the maximum value of the draws. We subtract the result with 1 and take the absolute value of this. This flips the distribution and we end up with a distribution that mimics a key distribution.

10 Experiments

The scope of the experiments for this project is to explore the Daisy Bloom Filter's memory footprint at a given false positive rate compared to other Learned Bloom Filters. To do so, we have implemented the Daisy Bloom Filter[12] along with the Partitioned Learned Bloom Filter, the Adaptive Learned Bloom Filter and the Regular Bloom Filter[13].

10.1 Setup

For the experiments, we have chosen the following Learned Bloom Filters to test: Regular Bloom Filter (Standard), Adaptive Learned Bloom Filter (Ada-BF), Partitioned Learned Bloom Filter (PLBF) and Daisy Bloom Filter (Daisy). In particular, we have chosen not to include the Disjoint Adaptive Bloom Filter. We have done so, since [7] shows that it has similar performance to the regular Adaptive Learned Bloom Filter. Furthermore, [18] shows that the Partitioned Learned Bloom Filter is strictly better than both in their experiments.

As the Daisy Bloom Filter is a theoretical paper with no prior implementations, we have made a series of assumptions to map the theoretical paper to code. One of the biggest challenges in mapping the theory to practice has been the notion that you in the Daisy draw n independent draws from the probability distribution $\mathcal{P}$ which results in a key set $\mathcal{P}_n$. This doesn't translate directly to practice, as these distributions are theoretical concepts. To provide an implementation, we have made the assumption, that $\mathcal{P}$ can be replaced with a machine learning model in practice. The ML model produces scores s, $0 < s < 1$. However, the scores are not in a scale that works for the proposed thresholds in the Daisy Bloom Filter paper. To solve this problem, we have normalized the scores.

[12] Our implementation is publicly available in: `https://github.com/BSc-learned-indexes/daisy-bf`

[13] The implementations of Partitioned Learned Bloom Filter, Adaptive Learned Bloom Filter and the Regular Bloom Filter are based upon work from the authors of [7] their implementation is provided here [8]

10.1.1 Generating p_x

To generate p_x for the Daisy Bloom Filter, we normalize the scores according to the equation below:

$$p_x = \frac{\text{score}(x)}{\sum_{i=1}^{n} \text{score}(i)} \tag{35}$$

10.1.2 Generating query distributions ($\mathcal{Q}$)

For the experiments, we have 3 different query distributions. First, we determine what the query probability q_x should be for a given element x.

Uniform query distribution

We set $q_x = \frac{1}{|U|}$, that is all elements are queried with the same probability.

Query distribution equal to $\mathcal{P}$

We set $q_x = p_x$, that is all elements are queried with the same probability as the probability of it being a key.

Query distribution as the inverse of $\mathcal{P}$

We set $q_x = 1 - p_x$, that is all elements are queried with the probability inverse to the probability of being a key. This flips the previous distribution.

We now have a value for q_x. To determine the number of times an element is queried in our experiments, we multiply each element's query probability q_x by a factor i. This value has been chosen to be 100,000. Since q_x is $0 < q_x < 1$ we query an element x at most 100,000 times and at least 1 time, as we want to make sure that all elements are queried at least once.

10.1.3 Details on the filters

The filters varies in how they're built. The Partitioned Learned Bloom Filter, Adaptive Learned Bloom Filter and the Regular Bloom are given an input size and then they build their respective Bloom Filter, which minimizes the false positive rate. The setup is inverted for the Daisy Bloom Filter. The Daisy Bloom Filter takes a target false positive rate, and returns the size of the bit-array that needs to be allocated to achieve the target false positive rate. All filters uses the MurMur3 [16] hash function in their implementation.

10.1.4 False positive probability target

When we test the Daisy Bloom Filter, we do so by generating a list of target false positive probabilities. We generate the list, $[f_0, ...f_{i-1}, f_i]$, by halving the input false positive probability f_0, i times:

$$f_i = \begin{cases} f_0 = f_0, \\ f_i = \frac{f_{i-1}}{2} \end{cases} \tag{36}$$

This is then used to build a Daisy Bloom Filter for each target false positive probability (Input parameter F in (32)). This is tested and the actual false positive rate is recorded.

10.1.5 Measuring the False Positive Rate

To measure the false positive rate of each implementation, we create a sequence of queries, where each element is a non-key and is queried $q_x \cdot i$ times according to a query distribution $\mathcal{Q}$. For each element we record the outcome of the query on the specific Bloom Filter. We calculate the false probability rate according to the following equation:

$$FPR = \frac{\sum_{x \in \mathcal{U} \setminus S} query(x) \cdot q_x \cdot i}{\sum_{x \in \mathcal{U} \setminus S} q_x \cdot i} \tag{37}$$

where $query(x)$ is defined as 0 or 1, depending on the outcome of the membership testing for the specific Bloom Filter:

$$query(x) = \begin{cases} 1 & \text{if } \texttt{Yes}, \\ 0 & \text{if } \texttt{No} \end{cases} \tag{38}$$

10.1.6 Finding optimal thresholds with τ

For one of the experiments we explore finding the optimal threshold using binary search. We do so by minimizing the difference between the input false positive probability, and the actual tested false positive rate.

10.2 Evaluation

To evaluate the performance of the Daisy Bloom Filter we have made different experiments exploring different scenarios it can be deployed in. There are quite a few variables to vary. These include the key and query distribution, the key to non-key ratio and different machine learning models. In the following experiments, we have isolated one variable and kept all others fixed to explore the effect the selected variable. Unless noted otherwise, the experiments on the URL data set have been conducted with a large Random Forest Classifier Model[14].

10.2.1 Experiment 1: Exploring different key to non-key ratios

Motivation

We know from Section 8.2 that different $\mathcal{P}$ and $\mathcal{Q}$ distributions, can effect the Daisy Bloom Filter's performance. One way to explore this is by changing the key to non-key ratio. We test the performance of the Bloom Filters with a query distribution which is inverse of $\mathcal{P}$. We do so because we expect the Daisy Bloom Filter to perform well with this query distribution. To compare this to the other learned approaches, we have set up 3 different experiments on 2 different data sets. Furthermore, we use the findings from this experiment to determine which key to non-key ratio to use in the rest of the experiments.

Parameters for the experiment

Experiment 1 is run with a a query rate which is inverse to the probability of being a key. This means that elements queried often, are unlikely to be keys, and elements queried rarely are likely keys.

[14] Details on size and performance of this model can be found in Section 10.2.4

Malicous URLs				Synthetic Zipfean			
Experiment	Keys	Non-keys	Query dist.	Experiment	Keys	Non-keys	Query dist.
Full key set	104,438	345,738	$1 - p_x$	Full key set	104,438	345,738	$1 - p_x$
1% keys	3457	345,738	$1 - p_x$	1% keys	4000	400,000	$1 - p_x$
0.1% keys	345	345,738	$1 - p_x$	0.1% keys	3000	3,000,000	$1 - p_x$

Table 2: Overview of parameters for Experiment 1.

Full key set

In this experiment we tested the Bloom Filters according to the key to non-key ratio of $\frac{104{,}438}{345{,}738} \approx 33\%$, i.e. one key to every 3 non-keys (the ratio of labels in the full URLs data set). The theory tells us that for Daisy to guarantee a false positive probability of at most F, then $\mathcal{P}$ and $\mathcal{Q}$ must satisfy $\sum_{x \in \mathcal{U}} p_x \cdot q_x \leq \frac{F}{n}$ as stated in section 8.2. For the URL data set the equation is approximately: $6.66 \cdot 10^{-8} \leq \frac{F}{104{,}438}$. This means that the Daisy Bloom Filter can only guarantee false positive rates of $104,438 \cdot 6.66 \cdot 10^{-8} \leq F$, which approximates to $0.0069 \leq F$. For the Synthetic data set this value is: $104,438 \cdot 1.65 \cdot 10^{-7} \leq F$, which means Daisy can only guarantee approximately: $0.017 \leq F$.

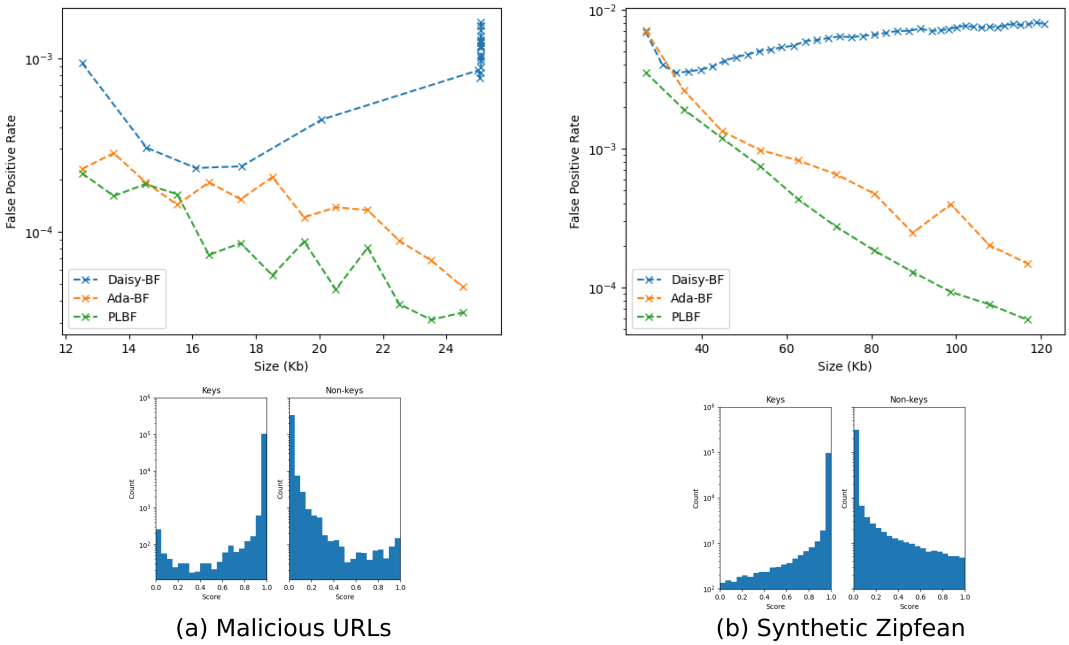


Figure 10: False positive rates on data sets with a key to non-key ratio of $\approx 33\%$.

In figure 10 we see that the false positive rate of the Daisy Bloom Filter decreases until a point where it starts to increase. This happens as the average number of hash functions for the keys increase, we can see this in the following Heat Map:

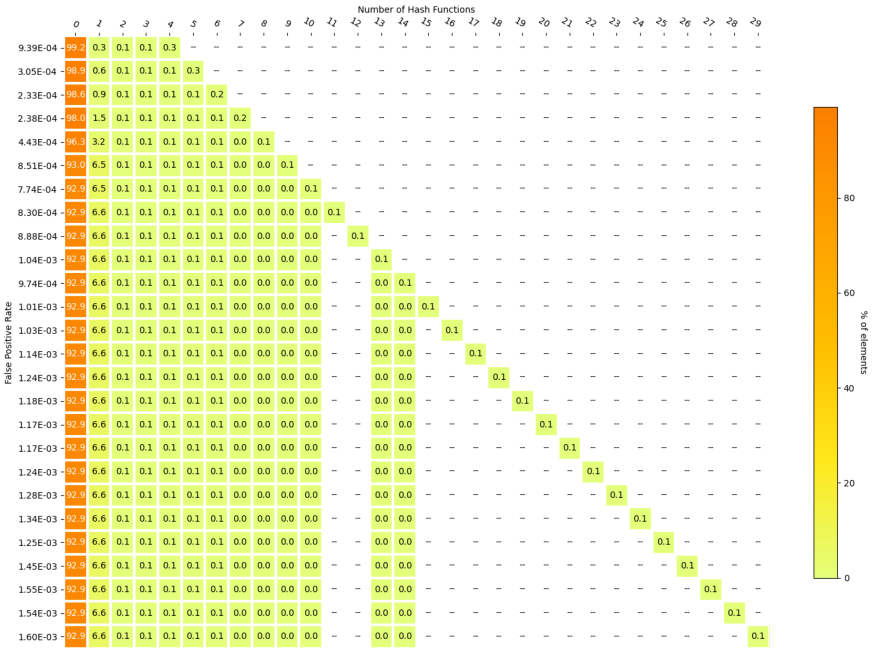
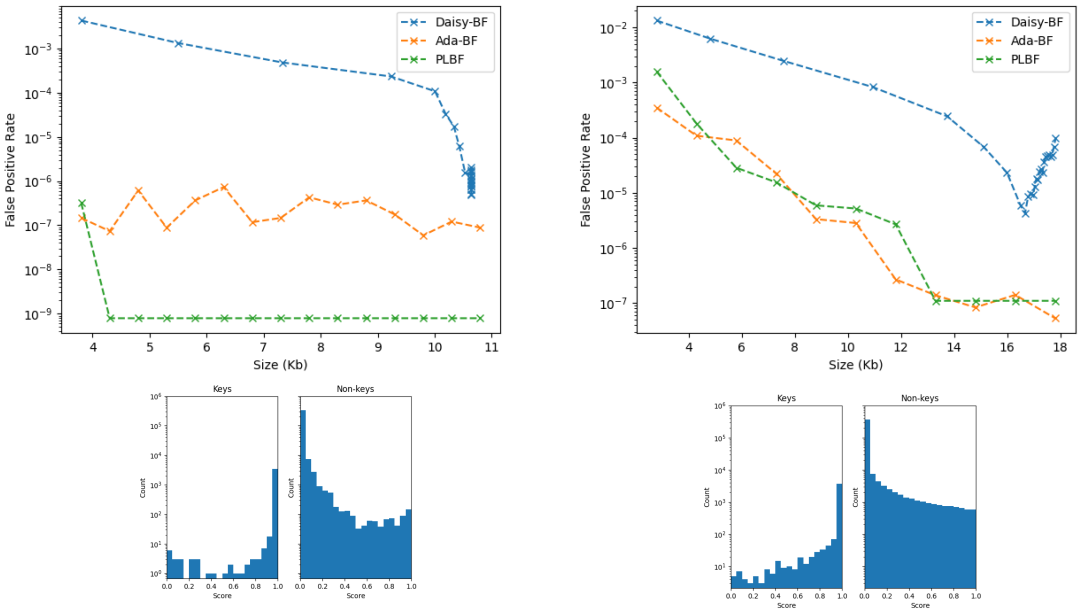


Figure 11: Heat Map of number of hash functions for inserted elements in the Daisy Bloom Filter on the URL data set.

Figure 11 shows that the amount of hash functions are increasing. Yet from Figure 10 (a) we see that the size of the bit array does not increase appropriately. This leads to more 1's being set in the bit array. The bit array becomes increasingly saturated. This increases the false positive rate. It is not the case for the other Bloom Filters. They keep decreasing the false positive rate within the allocated size.

1% key ratio

For this experiment we use a key ratio of 0.1%. We calculate the false positive guarantees for the Daisy Bloom Filter (29) for the URL data set as follows: $3457 \cdot 1.005 \cdot 10^{-6} \leq F$ which approximates to $0.0034 \leq F$. For the Synthetic Zipfean data set we get the following: $4000 \cdot 9.51 \cdot 10^{-7} \leq F$ which approximates to $0.0038 \leq F$.



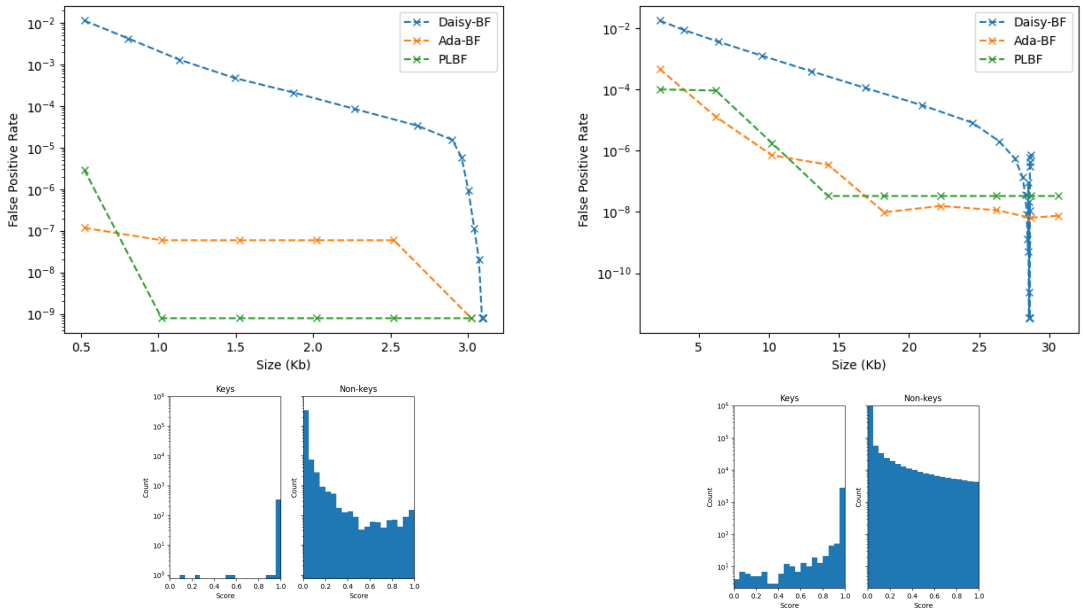
(a) Malicious URLs (b) Synthetic Zipfean

Figure 12: False positive rates on data sets with a key to non-key ratio of 1%.

In Figure 12 (a) the Daisy Filter approaches the false positive rate of the Adaptive Learned Bloom Filter, but then stalls. In 12 (b) the false positive rate of the other Learned Bloom Filters is lower. In both data sets we initially see a decrease in the false positive rate of the Daisy Bloom Filters. This is followed by an increase of the false positive rate at the size of 10.6 Kb in (a) and (b) 17 Kb. The same problem as discussed in Full key set experiment happens. The bit array becomes over saturated with 1's, due to the face that the size allocation does not scale appropriately. In (a) the Partitioned Learned Bloom Filter reaches the limit of the classification model. All the false positives from the size of 4 Kb and onwards comes from the model. Most of the false positives from the Adaptive Learned Bloom Filter comes from the underlying Bloom Filter and not the model itself. In (b) the Partitioned Learned Bloom Filter reaches the limit of the model at 13.3 Kb. The Adaptive Learned Bloom Filters haven't yet reached the limit of the model.

0.1% key ratio

For this experiment we use a key ratio of 0.1%. For Daisy to satisfy (29) for the URL data set we must satisfy: $345 \cdot 1.96 \cdot 10^{-6} \leq F$ which approximates to $0.00067 \leq F$. However, note that sampling only 345 elements of the key set can lead to drastically different results depending on which elements are sampled. For the Synthetic Zipfean data set we get the following: $3000 \cdot 1.65 \cdot 10^{-7} \leq F$ which approximates to $0.00049 \leq F$.



(a) Malicious URLs (b) Synthetic Zipfean

Figure 13: False positive rates on data sets with a key to non-key ratio of 0.1%.

In Figure 13 (a) there is a steep decrease in the false positive rate at a size of 2.9 Kb. This is also seen on (b) at a size of 28 Kb. In (a) the Partitioned Learned Bloom Filter reaches the limit of the model at 1 Kb, while the Daisy and Adaptive Learned Bloom Filter does so at 3 Kb. For (b) the Partitioned Learned Bloom Filter there is a constant 9688 false positives from the model. The Adaptive Bloom Filter suffers from the same problem, but with 962 false positives. It might be possible to lower this amount with a better optimization of the thresholds for both the Partitioned and Adaptive Learned Bloom Filters. The Daisy is able to reach a point where only a single query element is a false positive.

Findings

From the 3 key ratio experiments we see the performance of the Bloom Filters rely on the key ratios. This is especially apparent for the Daisy Bloom Filter. Furthermore, we see that the Daisy Bloom Filter is performing worse at most sizes. We also see that for a key ratio of 0.1% there is a narrow range of sizes where the Daisy performs just as well or better than the other learned models.
Moving forward we will use 0.1% keys for the Zipfean data set. For the URL data set, we will use 1% due to the low amount of keys.

10.2.2 Experiment 2: A uniform query distribution

Motivation

In this experiment, we evaluate the performance of the Learned Bloom Filters under the conditions that are implicit assumed in [7] [18]. Namely, the the elements are queried uniformly. This experiment shows how the Daisy Bloom Filter performs in a setting that it is not designed for. Furthermore, we attempt to optimize the Daisy Bloom Filter by binary searching for other thresholds.

Parameters for the experiment

To see how the Daisy Bloom Filter compares to the Partitioned and Adaptive Learned Bloom Filters we have tested them on the Malicious URLs with 1% key ratio and the Synthetic Zipfean data set with 0.1% key ratio. We query each non-key once, as the query distribution is uniform.

Malicous URLs				**Synthetic Zipfean**			
Experiment	Keys	Non-keys	Query dist.	Experiment	Keys	Non-keys	Query dist.
1% keys	3457	345,738	Uniform	0,1% keys	3000	3,000,000	Uniform

Table 3: Overview of parameters for Experiment 2.

Experiment 2a: Uniform query distribution

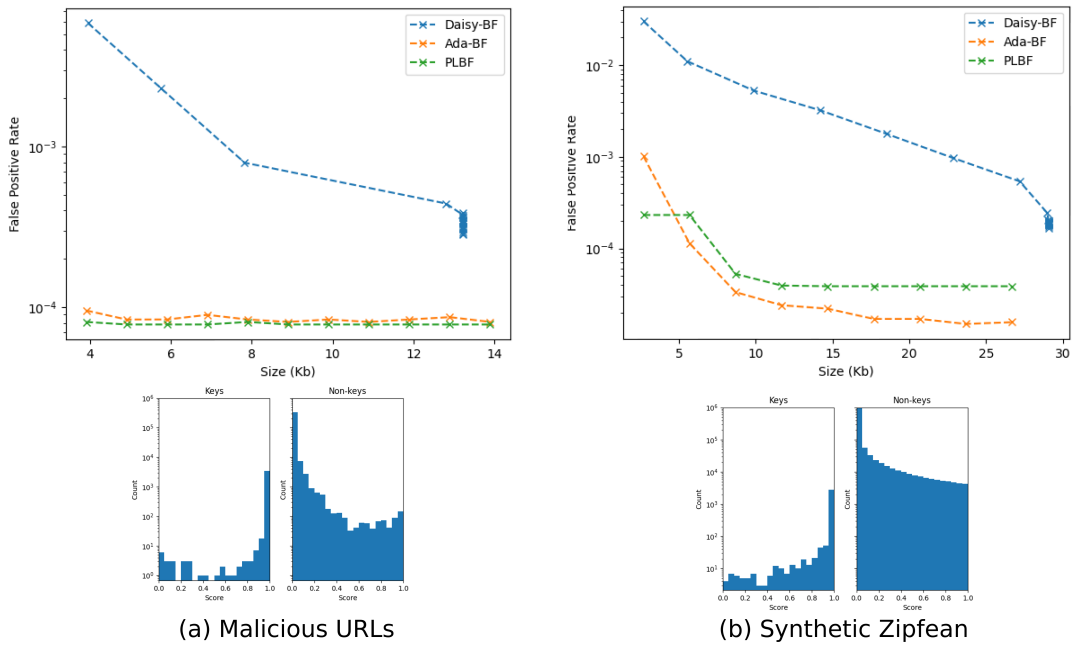


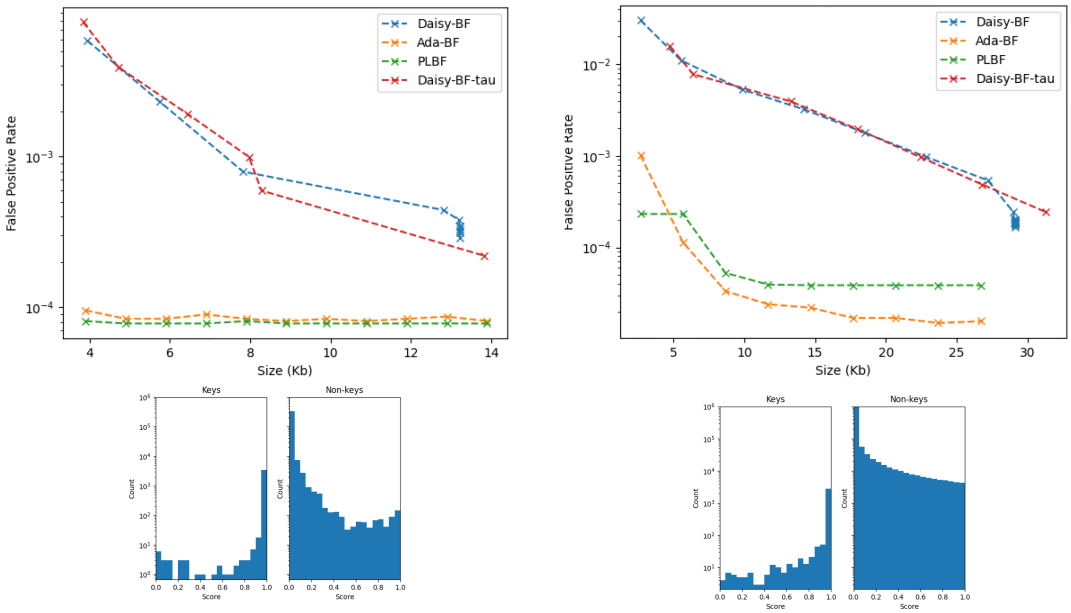
Figure 14: False positive rates for each Learned Bloom Filter with an uniform query distribution.

In Figure 14 (a) we see that the performance of all the filters is worse than in Figure 12 (a). The Partitioned and Adaptive Learned Bloom Filters are performing well as they reach the limit of the

classification model.
The Daisy Bloom Filter reduces the number of false positives from the model to 0 at a size of 12.8 Kb[15]. The Daisy Bloom Filter struggles to reduce the false positives from the underlying Bloom Filter. It suffers from the same problem again. It saturates the bit array by using more hash functions without increasing the size appropriately.
The same things happens for (b). However, one difference is that the thresholds chosen for the Adaptive Learned Bloom Filter reaches the limit of its classifier. It repeatably has 34 false positive items in contrast to 116 for the Partitioned Learned Bloom Filter.
The Daisy Bloom Filter is performing a lot worse compared to the other Bloom Filters as it struggles to reach the same false positive rate. This is in contrast to Figure 13 (b) where the best performing filter at the size of 28 Kb was the Daisy.

Experiment 2b: Binary searching for τ

For this experiment we keep the same conditions as before. We try to improve the Daisy Bloom Filter by binary searching for the optimal thresholds. We do so by picking the thresholds that give an actual false positive rate closest to the target false positive rate. In the following Figure, we see similar results when optimizing the thresholds as the regular Daisy Bloom Filter.



(a) Malicious URLs (b) Synthetic Zipfean

Figure 15: False Positive Rates for each Learned Bloom Filter, with an uniform query distribution and binary search for τ.

[15]This can be seen in row 4 of the Heat Map for inserts in Appendix A.1

Findings

We see that the Daisy Bloom Filter performs worse with a uniform query distribution in comparison to the other Learned Bloom Filters and other query distributions. Furthermore, we see that there is no significant difference between the theoretical thresholds and binary searching the optimal threshold values.

10.2.3 Experiment 3: A query distribution equal to the key distribution

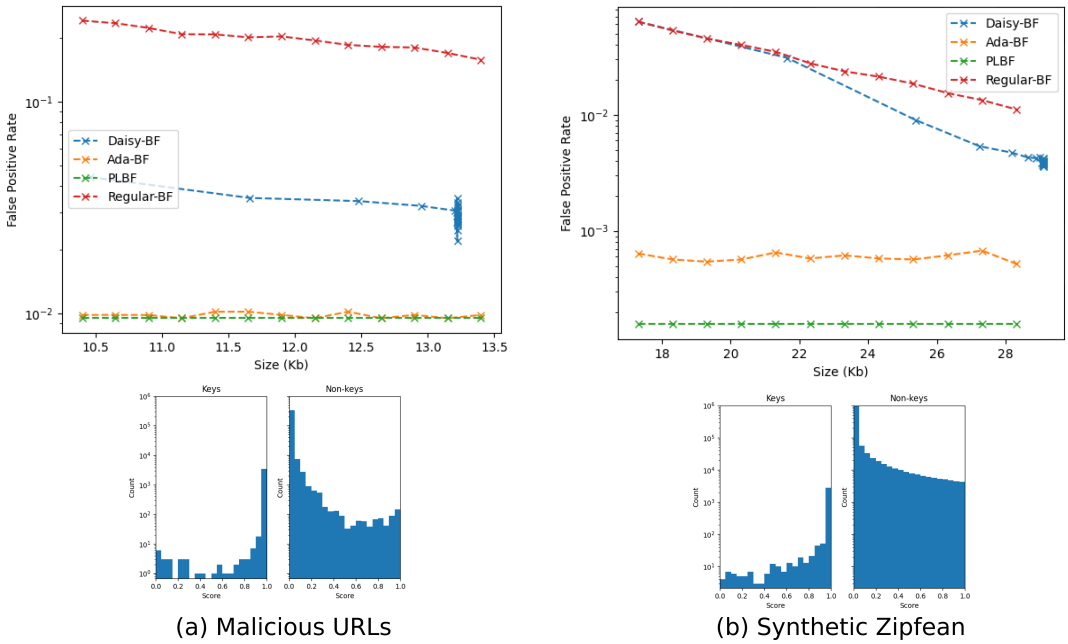
Motivation

The authors of the Daisy Bloom Filter[2] claim that the performance cannot get better than the Regular Bloom Filter under the condition that $q_x = px$. This would imply that all elements are hashed $\log\left(\frac{1}{F}\right)$ times, which means that they all end up in the gray area on Figure 7. We explore this in this experiment.

Parameters for the experiment

Malicous URLs				Synthetic Zipfean			
Experiment	Keys	Non-keys	Query dist.	Experiment	Keys	Non-keys	Query dist.
1% keys	3457	345,738	$q_x = p_x$	0,1% keys	3000	3,000,000	$q_x = p_x$

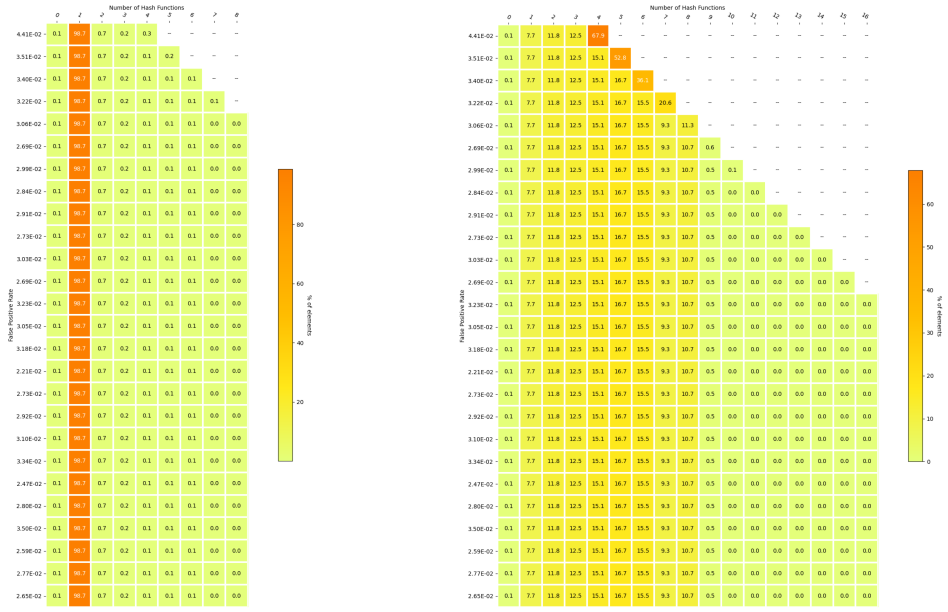
Table 4: Overview of parameters for Experiment 3.



(a) Malicious URLs (b) Synthetic Zipfean

Figure 16: False positive rates for each Learned Bloom Filter with a query distribution of $p_x = q_x$.

In Figure 16 we see that Daisy Bloom Filter performs better than the Regular Bloom Filter in both (a) and (b). This indicates, that the filter hashes some of the element less than $\log\left(\frac{1}{F}\right)$ times. Otherwise we would see similar performance as the Standard Bloom Filter. To investigate this further, we provide the following Heat Maps:



(a) Daisy Bloom Filter, Inserts

(b) Daisy Bloom Filter, Lookups

Figure 17: Heat Maps of number of hash functions in Daisy Bloom Filter on the URL data set.

In Figure 17 (b)[16] the rightmost cell in each row corresponds to how many elements are hashed $\log\left(\frac{1}{F}\right)$ times. All elements in between that and the first column comes from elements that land in $\mathcal{U}_2 \cup \mathcal{U}_4$ and are determined by the thresholds described in Section 8. What we notice here, is that the filter does not behave as expected. If it were to behave as theoretically described, we would expect to see one cell with a value of 100 for each row. The exact same insights can be seen for the Zipfean Data set that is portrayed in Figure 16 (b). The heat map for this is provided in Appendix A.2.
Again we see that the Daisy Bloom Filter doesn't increase its size, thus the false positive rate stalls.

Findings

The Daisy Filter performs better than expected given this query distribution. We see this in Figure 16 where the Daisy Bloom Filter has a lower False Positive Rate than the Regular Bloom Filter given the same space.

[16]Larger illustrations are provided in Appendix A.2.

10.2.4 Experiment 4: Various machine learning models on URL set

Motivation

The machine learning models are a key component in Learned Bloom Filters. In each Learned Bloom Filter assume that the model is able to accurately score each element. Thus, the performance of the model, may impact the Learned Bloom Filters.

Parameters for the experiment

Malicous URLs					
Experiment	Keys	Non-keys	Query dist.	Max. leaf nodes	N. estimators
Large RFC	3457	345,738	$1 - p_x$	None	100
Small RFC	3457	345,738	$1 - p_x$	20	10
Logistic Regression	3457	345,738	$1 - p_x$	-	-

Table 5: Overview of parameters for Experiment 4.

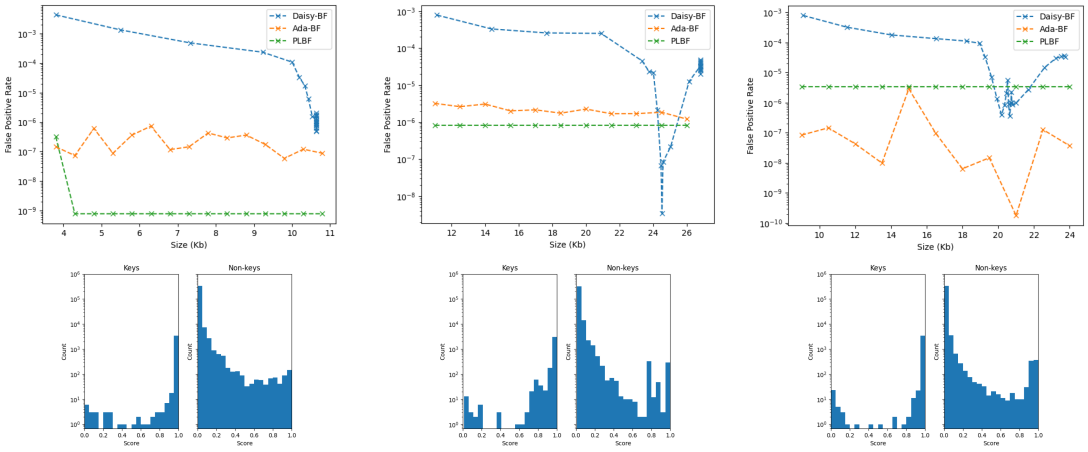
To evaluate the impact of different models, we have tested 3 variations of machine learning models. A Random Forest Classifier, which is using the default parameters from Scikit-learn [14]. A Random Forest Classifier with 10 trees, each having at maximum 20 leaf nodes. At last we test a Logistic Regression model, which uses the default parameters of the Scikit-learn implementation [15].

Model performance indicators

Model	Size (bits)	Accuracy	F1-score	Precision	FP
Large RFC	73,367,472	0.995595	0.995577	0.99804469	142
Small RFC	261,112	0.995362	0.995342	0.998112401	137
Logistic Regression	9088	0.994042	0.994009	0.9978322701	157

Table 6: Metrics of different machine learning models.

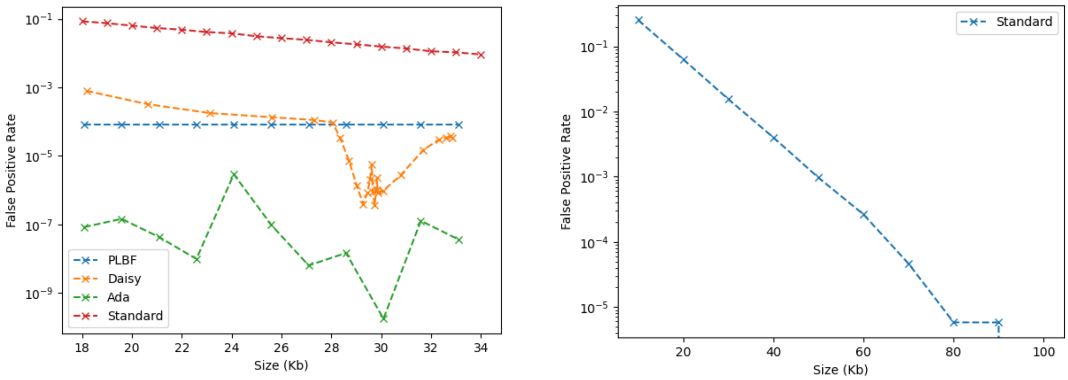
The performance of these models can be seen in Table 6. All the models have similar accuracy, F1-score and precision. We see that the model that produces the least amount of false positives is the Small RFC.



(a) Random Forest Classifier, large (b) Random Forest Classifier, small (c) Logistic Regression

Figure 18: Performance of filters with different machine learning models.

In Figure 18 we see that all the filters perform well with the Large Random Forest Classifier (a). Furthermore, we see that the Adaptive and Partition Learned Bloom Filter are performing strictly worse with the Small Random Forest Classifier (b). For the Daisy Bloom Filter, we see that it manages to perform well on this model at a particular point around 24 Kb. The Adaptive Learned Bloom Filter performs best on the Logistic Regression model (c).



(a) Learned Bloom Filters and Regular (b) Regular Bloom Filter

Figure 19: Total size of Learned Bloom Filters with 9 Kb model and the Regular Bloom Filter.

In Figure 19 (a) each Bloom Filter is plotted with the total size. All values for the Learned Bloom Filters are the same as Figure 18 (c) + the size of the model (9088 bits). Even though we have added the size of the model, the Learned Filters all have a lower false positive rate than the regular Bloom Filter.
For Learned Bloom Filters to be feasible, the model size in itself has to be smaller than the size of the regular Bloom Filter. As we can see in Table 6, the size of the both of the Random Forest Classifiers are too large. That is, they are both larger than the size that the regular Bloom Filter needs to allocate in order to reach a false positive rate of 0. That size can be seen in in Figure 19 (b). This means that we cannot use these two models in practice if take the cost of the model into consideration.

Findings

From the experiment we see that the performance of the different Learned Bloom Filters depend on the machine learning model. If you take the total space occupied by a Bloom Filter into account, there is a case to be made to stick to the Regular Bloom Filter in certain conditions. This could be if the machine learning model is large in it's memory usage. In such situations, we see that the Regular Bloom Filter might reach a smaller false positive rate using less space, than Learned Bloom Filters with a large model. Thus, the setting of which the Bloom Filters are deployed can dictate the optimal Bloom Filter. Another thing to note is that the Daisy Bloom Filter might require two machine learning models, one to produce $\mathcal{P}$ and another to produce $\mathcal{Q}$. This could drastically increase the amount of space needed for the Daisy Bloom Filter.

11 Conclusion

This project has explored Learned Bloom Filters and particularly the Daisy Bloom Filter. It has provided the first implementation of the filter and compared it's performance to two state-of-the-art Learned Bloom Filters: Adaptive Learned Bloom Filter and Partitioned Learned Bloom Filter as well as the Regular Bloom Filter. It has been done in a setting that is open source and reproducible. Under the conditions in the various experiments performed in this project, we see that the Daisy Bloom Filter in most cases performs worse than the other learned approaches. However, it has also show that the results depend heavily on the conditions on which the experiment is performed. It has shown that by providing specific combinations of key distributions, query distributions, key to non-key ratios and machine learning models, it is possible to get vastly different results between the filters. Furthermore, it has shown that all the Learned Bloom Filters in our setups perform better than the Regular Bloom Filter. That is, if we do not include the size of the machine learning models. This provides many interesting opportunities for memory optimization in practical applications.

11.1 Further work

In this project we have focused on exploring the false positive rate and memory performance of the Daisy Bloom Filter. One thing to further explore is using different query distributions, as the experiments presented have used query distributions which where either dependent on $\mathcal{P}$ or uniform. Other query distributions might produce other results. Another exiting direction of study would be to benchmark the time complexity of the Daisy Bloom Filters insert, query and build time, compared to the other Bloom Filters. In addition to this, another interesting thing to investigate would be how the different Learned Bloom Filter's performance are affected by additional inserts after the initial creation of the filters.

References

[1] Aleksandra Bartosik and Hannes Whittingham. "Chapter 7 - Evaluating safety and toxicity". In: *The Era of Artificial Intelligence, Machine Learning, and Data Science in the Pharmaceutical Industry*. Ed. by Stephanie Kay Ashenden. Academic Press, 2021, pp. 119–137. ISBN: 978-0-12-820045-2. DOI: https://doi.org/10.1016/B978-0-12-820045-2.00008-8. URL: https://www.sciencedirect.com/science/article/pii/B9780128200452000088.

[2] Ioana O. Bercea, Jakob Bæk Tejs Houen, and Rasmus Pagh. *Daisy Bloom Filters*. 2022. DOI: 10.48550/ARXIV.2205.14894. URL: https://arxiv.org/abs/2205.14894.

[3] Burton H. Bloom. "Space/Time Trade-Offs in Hash Coding with Allowable Errors". In: *Commun. ACM* 13.7 (July 1970), pp. 422–426. ISSN: 0001-0782. DOI: 10.1145/362686.362692. URL: https://doi.org/10.1145/362686.362692.

[4] Andrei Broder and Michael Mitzenmacher. "Network applications of bloom filters: A survey". In: *Internet mathematics* 1.4 (2004), pp. 485–509. URL: https://www.eecs.harvard.edu/~michaelm/postscripts/im2005b.pdf.

[5] Chabacano. *Overfitting*. Apr. 2023. URL: https://en.wikipedia.org/wiki/Overfitting (visited on 05/15/2023).

[6] Chromium. *bloom_filter.cc*. 2023. URL: https://chromium.googlesource.com/chromium/chromium/+/refs/heads/main/chrome/browser/safe_browsing/bloom_filter.cc (visited on 05/15/2023).

[7] Zhenwei Dai and Anshumali Shrivastava. *Adaptive Learned Bloom Filter (Ada-BF): Efficient Utilization of the Classifier*. 2019. DOI: 10.48550/ARXIV.1910.09131. URL: https://arxiv.org/abs/1910.09131.

[8] DAIZHENWEI. *Ada-BF*. 2023. URL: https://github.com/DAIZHENWEI/Ada-BF (visited on 05/15/2023).

[9] Tim Kraska et al. *The Case for Learned Index Structures*. 2018. arXiv: 1712.01208 [cs.DB].

[10] Siddharth Kumar. *Detect Malicious URL using ML*. 2023. URL: https://www.kaggle.com/code/siddharthkumar25/detect-malicious-url-using-ml (visited on 05/15/2023).

[11] Michael Mitzenmacher. *A Model for Learned Bloom Filters, and Optimizing by Sandwiching*. 2019. DOI: 10.48550/ARXIV.1901.00902. URL: https://arxiv.org/abs/1901.00902.

[12] Scikit-learn. *1.10.7. decision trees: Mathematical formulation*. URL: https://scikit-learn.org/stable/modules/tree.html#mathematical-formulation (visited on 05/15/2023).

[13] Scikit-learn. *1.11.2.3. parameters*. URL: https://scikit-learn.org/stable/modules/ensemble.html#parameters (visited on 05/15/2023).

[14] Scikit-learn. *Sklearn.ensemble.randomforestclassifier*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html (visited on 05/15/2023).

[15] Scikit-learn. *sklearn.linear_model.LogisticRegression*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html (visited on 05/15/2023).

[16] Scikit-learn. *sklearn.utils.murmurhash3_32*. 2023. URL: https://scikit-learn.org/stable/modules/generated/sklearn.utils.murmurhash3_32.html (visited on 05/15/2023).

[17] A. Aylin Tokuç. *Underfitting and overfitting in machine learning*. Nov. 2022. URL: https://www.baeldung.com/cs/ml-underfitting-overfitting (visited on 05/15/2023).

[18] Kapil Vaidya et al. *Partitioned Learned Bloom Filter*. 2020. DOI: 10.48550/ARXIV.2006.03176. URL: https://arxiv.org/abs/2006.03176.

[19] George Kingsley Zipf. *The psycho-biology of language: An introduction to dynamic philology*. Vol. 21. Psychology Press, 1936.

A Experiments

A.1 Experiment 2

A.1.1 Heat Maps: URL data set

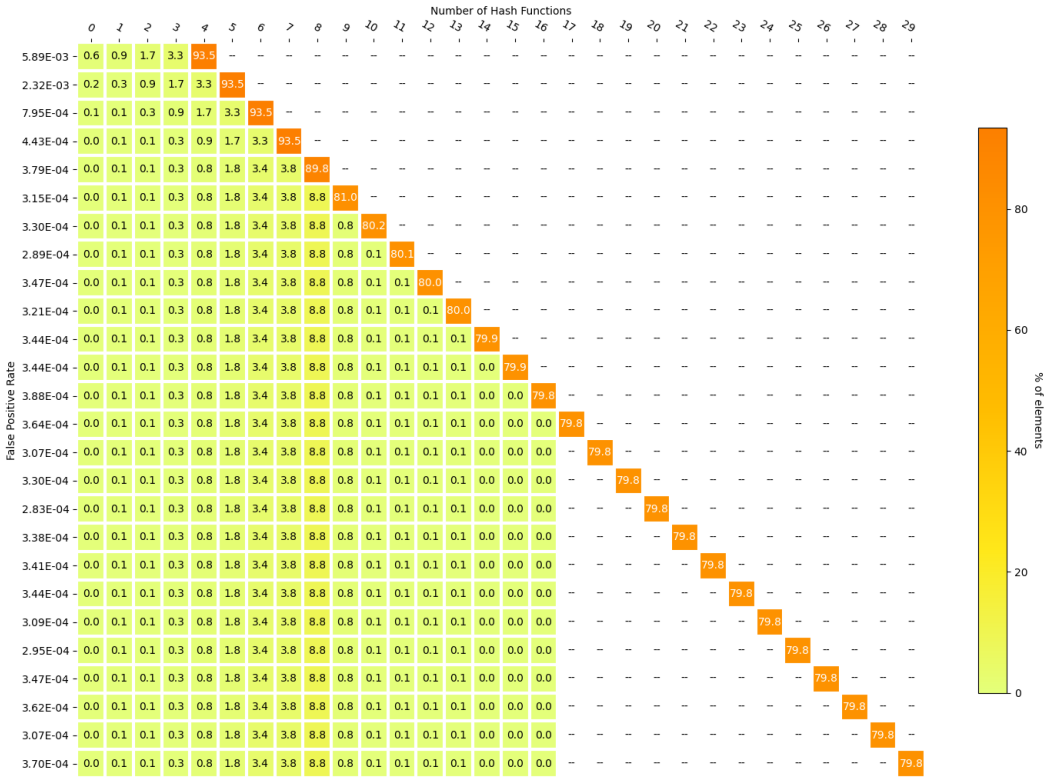


Figure 20: Experiment 2a: Heatmap lookups URL data set .

A.2 Experiment 3

A.2.1 Heat Maps: URL data set

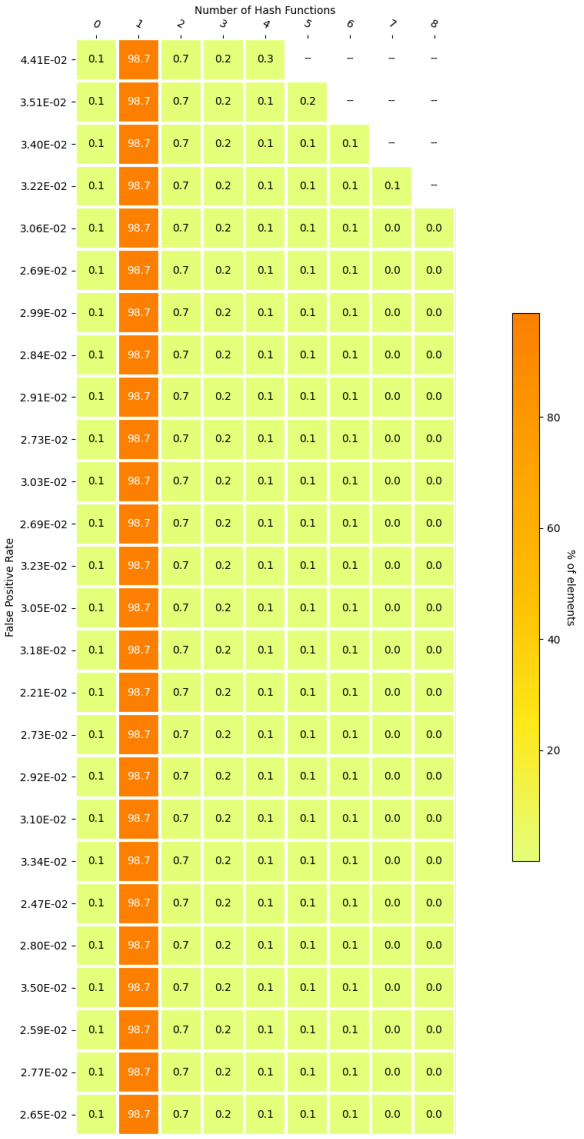


Figure 21: Heatmap inserts URL data set.

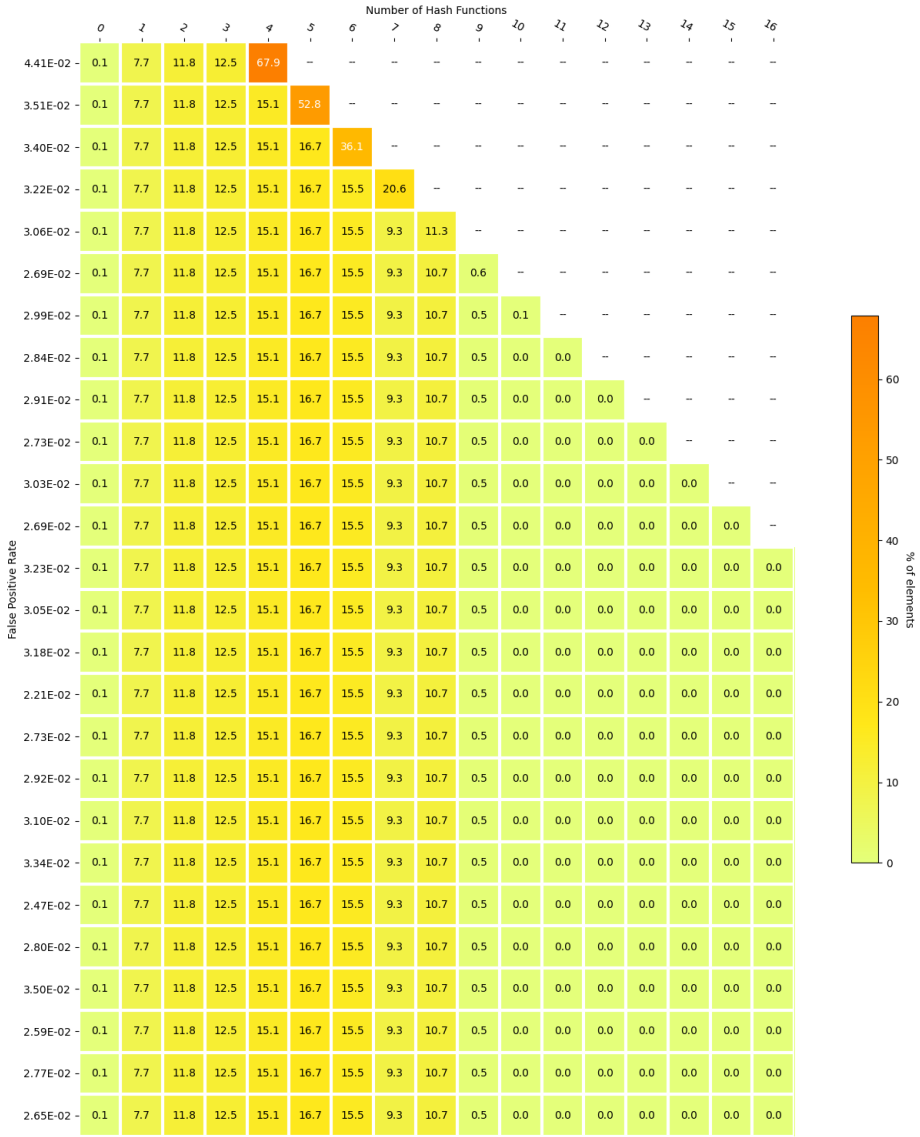


Figure 22: Heatmap lookups URL data set.

A.2.2 Heat Maps: Zipfean data set

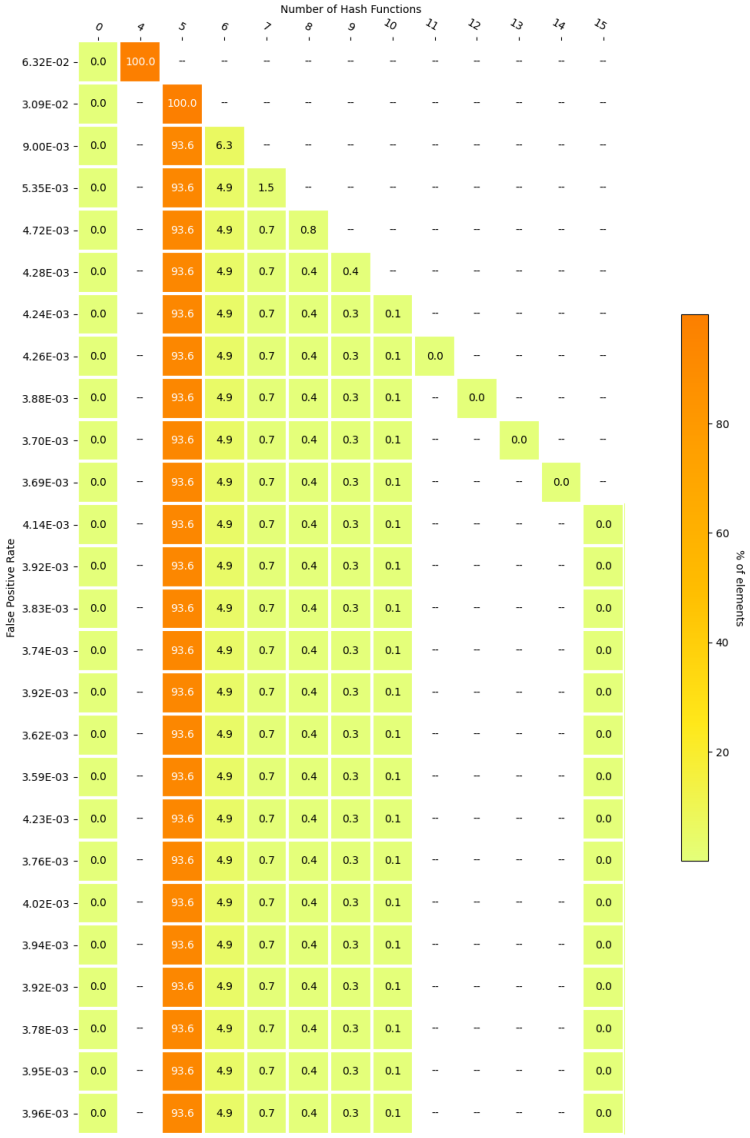


Figure 23: Heatmap inserts Synthetic Zipfean data set.

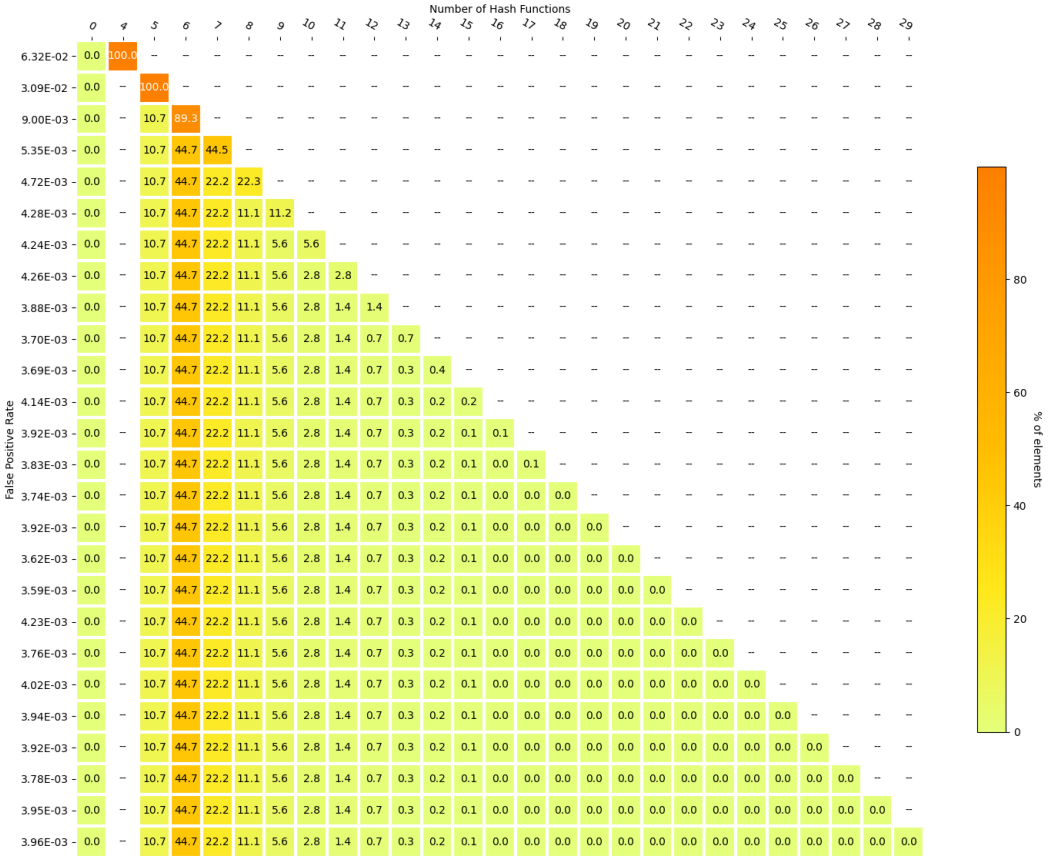


Figure 24: Heatmap lookups Synthetic Zipfean data set.