Learned Bloom Filters

John Yang

Princeton University — COS 513 Final Project

Email: jy1682@princeton.edu

Abstract. This paper explores the thread of research applying machine learning techniques to bloom filters. In this project report, we describe the designs and implementations of the naive, learned, sandwiched, adaptive, and disjoint-adaptive bloom filters. The filters are then evaluated on a dataset to quantify the performance benefits of such learned data structures. Finally, we discuss how different hyper-parameters affect how optimal performance is achieved.

1 Introduction

The Bloom Filter is a widely used probabilistic data structure for performing approximate membership testing. Given a set of items, Bloom Filters run the values through a fixed number of hashing functions, then aggregates and saves the outputs into a bit array. Compared to alternatives such as Tries and Hash Tables, Bloom Filters are usually preferred in larger systems due to their low memory usage and fast query time [1]. Bloom Filters are used for a wide variety of applications, including web caching, anomaly detection, address lookup, and network telemetry [2].

In 2017, the Google Brain team proposed the idea of learned index structures, which purports an approach of applying machine learning techniques to indexing and membership tasks normally performed by traditional data structures [3]. The paper argues that the inherent problem being solved is modeling the underlying cumulative distribution function of the target dataset. While the heuristics of traditional structures are unaware of such signals, ML techniques can not only consider but also act upon such information. Along these lines, the authors train a binary classifier and bloom filter hybrid model, dubbed a "learned bloom filter", that outperformed naive bloom filters with both lower memory usage and greater accuracy.

A variety of research directions towards applying learning techniques to traditional data structures problems proliferated after this paper. A specific thread of this research features re-designs of this learned bloom filter model, each of which have led to higher accuracy while also minimizing storage costs. This paper collectively presents and benchmarks these learned bloom filters — naive, learned, adaptive, disjoint-adaptive, and sandwiched bloom filters — in one place.

2 Overview

This section presents brief summaries of the bloom filter and its learned variants. After section 2.1, the subsequent sections describe how that particular bloom filter builds upon the prior's original design.

2.1 Naive

An empty bloom filter is a bit array with m slots all initialized to 0. To insert an element into the bloom filter, the element is fed through k independent hash functions, producing a set of k integers that are used as positions. The corresponding positions in the array are set to one. To query whether an element exists, the element is hashed and the output positions are checked. If all positions have a value of one, then the element exists. Otherwise, it does not.

When queried, bloom filters may produce false positives, but will never produce false negatives, which translates to one of two results — *definitely does not exist* or *possibly exists* [4]. The false positive rate, which is the main accuracy metric, is directly affected by the ratio of hash function k to size of the bit array m; a ratio of 0.693 is formally proven as ideal [5].

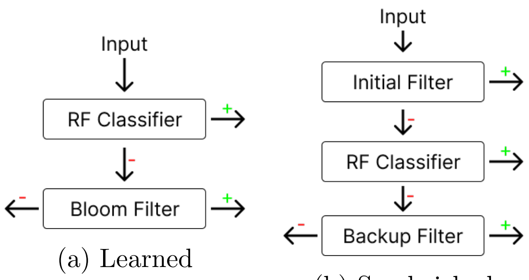
2.2 Learned

The original argument behind the learned bloom filter design is that determining existence can be reframed as a classification problem [3]. Based on the classifier's confidence in the prediction, the key is said to exist if it's above a confidence threshold. Otherwise, a backup bloom filter is used to handle cases with lower confidence scores. This design divides the target data distribution into two halves; the half handled by the learned model boasts significant memory footprint reduction and inference speed gains, while also ensuring the original guarantee of no false negative results [2].

For higher thresholds, the classifier's prediction yields fewer false positives, but the backup bloom filter inserts more data and is queried more often. In combination, this leads to improved accuracy but sub-optimal storage efficiency. On the other hand, a lower threshold means fewer queries are sent to the backup bloom filter, but the classifier is more likelihood to produce errant predictions. This balance is why, during the training process, an optimal threshold specific to the dataset is sought out via training across a range of candidate thresholds.

2.3 Sandwiched

The responsibility of the bloom filter in the learned bloom filter is to eliminate the possibility of false negatives that might be caused by the learned model [2]. The observation made by the author of the sandwiched bloom filter is to reduce the number of tricky false negatives that the learned model would potentially mis-classify. The sandwiched bloom filter model places an *initial filter* in front of the learned model; unlike the traditional filter, this initial filter's responsibility is to definitively indicate if a key does *not* exist. Otherwise, it forwards the key to the learned model. The placement of the learned function between two bloom filters is what leads to the "sandwiched" namesake.



(a) Learned

(b) Sandwiched

Figure 1: Visualizations of the learned and sandwiched bloom filters. The + and - refer to the positive and negative URLs, respectively.

2.4 Adaptive

The LBF splits a data distribution into two groups, with the existence problem respectively handled by either zero or k hash functions. The adaptive bloom filter generalizes this idea to dividing a region into g groups, where each group is bucketed by a range of confidence thresholds; a specifically tuned number of independent hash functions are used for each group [6]. The keys across all groups are hashed into a shared, naive bloom filter. This greater degree of flexibility allows the adaptive learned bloom filter to mimic the overall distribution and the granularity of each region more closely. During training, the number of groups and hash functions for each group are chosen via a naive grid search aimed at minimizing the macro-average FPR across all groups.

2.5 Disjoint-Adaptive

The hashed keys across all g groups of the adaptive bloom filter are fed into a shared naive Bloom Filter. The disjoint model proposes hashing keys from separate groups into separate bloom filters. This disparity in design affects how the FPR of each group is influenced; while the adaptive model updates the value of k (number of hash functions), the disjoint model instead changes the bucket allocation. In general, both models attempt to lower the average FPR by reducing the FPR of groups with more non-keys.

3 Implementation

This section is divided into two parts. In part 3.1, I discuss the technical details behind the implementations of each of the bloom filters discussed 1. In part 3.2, I cover how I put together a Random Forest classifier for predicting whether a URL is malicious or benign. This classifier is used to generate the confidence scores that the learned bloom filters uses to bucket values and determine the appropriate number of hash functions based on the range of thresholds each bucket covers.

3.1 Bloom Filters

All bloom filters share the same class definition in addition to some parameters for configuring the bloom filter. Each bloom filter implements two main methods: *insert* and *query*.

Generally, the insert query takes a list of URLs and stores the hashes of the URLs into the underlying bloom filter(s) accordingly. The *query* function takes in an individual URL and returns a label of either "1" or "-1" to indicate that the URL does or does not exist, respectively. To determine the number of slots that each bloom filter has, each bloom filter takes in a *size_of_BF* argument that accepts an integer input value corresponding to the number of slots. For the evaluation mechanism, the benign queries are added to the bloom filter via the *insert* function. Then, the malicious queries are passed in via the *query* function. Since bloom filters do not produce false negatives, we only need to evaluate the false positive rate with the malicious queries to get an accurate reflection of the bloom filter's performance.

The naive bloom filter uses a *murmurhash*3 function. The naive implementation is self contained. The learned bloom filters import and use the naive design in their implementations. Generally, each of the learned bloom filters take in arguments corresponding to a set of ranges that are then each searched across to determine the optimal settings for the learned bloom filter on the target dataset. The learned and sandwiched bloom filters search across a range defined by *threshold_min* and *threshold_max* command line arguments in intervals of 0.05. The purpose of this search is to determine what threshold is optimal for splitting the task of determining existence between the classifier (discussed in 3) and backup bloom filter(s). In addition to these arguments, the adaptive and disjoint-adaptive bloom filters also take in a *c_min* and *c_max* argument, which corresponds to a range of key ratios that is searched across to determine the appropriate number of hash functions per bucket.

3.2 Dataset

The main task used to train and evaluate the Bloom Filters is to identify malicious URLs, retrieved from Kaggle. The raw dataset contains 800K+ URLs in their raw format, all of which are labeled with either "-1" or "1" to indicate either a malicious or benign URL. The dataset is split into 46% malicious URLs and 54% benign URLs. There are no redundancies in the dataset, although it's possible that URLs with the same domain but different paths are used.

3.3 Classifier

The core part of the learned bloom filter is a machine learning model that is trained to determine existence. For the target URL dataset used for evaluation 3, a Random Forest model was trained on a set of 27 features, with the number of estimators set to 100. The 27 features consist of a variety of handcrafted features, such as the length of the query, the presence of suspicious words or punctuation in the URL, and dubious patterns. The *sklearn.ensemble*'s module was used to fit and tune the dataset on a Random Forest model. The model was trained and tested on a 70/30 split. K-fold cross validation was used to determine accuracy, producing an average of 0.92 classification accuracy. Given the 60/40 distribution of benign and malicious URLs, this can be considered a fairly strong classifier.

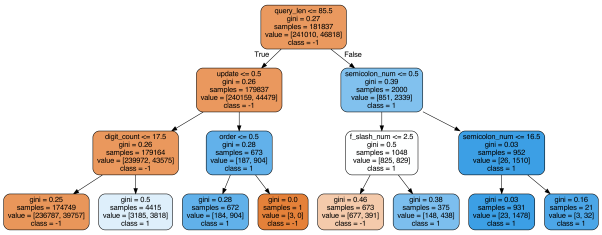


Figure 2: Random Forest Visualization

The tree visualization in 2 provides insight into what factors are most telling for the output Random Forest model; the length of a query, along with the presence of specific words and punctuation seem to be most telling.

Once the random forest model was fixed, the model was used to determine the scores for all of the URLs using the prediction function. Figure 3 depicts two histograms that depicts the counts of the scores for the malicious and benign URLs. The nearly mirror image distributions for the malicious and benign URLs indicate that such a classifier would be able to learn a highly accurate decision boundary between the two categorizes. In turn, this means that not only should the false positive rate decrease, but the resulting backup bloom filters that are used for each of the learned models should be of a much smaller size relative to the naive version.

4 Evaluation

Two sets of experiments were carried out. First, the general false positive rates of each of the bloom filters are compared via the URL

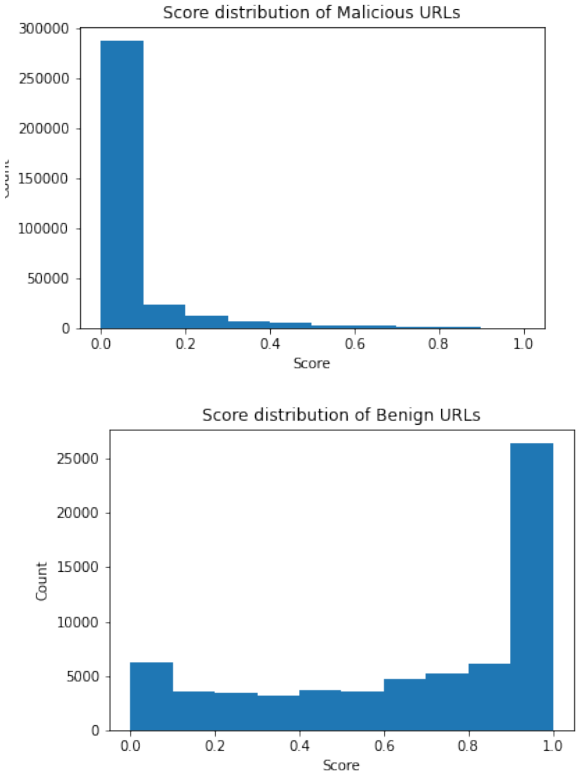


Figure 3: Histograms of score distributions of keys (Malicious) and non-keys (Benign) URLs.

dataset. The absolute numbers and the trend lines of the FPR rates confirm the effects of specific designs for different learned bloom filters while also providing insight into how different aspects of a dataset's distribution may affect the disparity in performance between the learned bloom filters. The second section delves into how the optimal values for hyperparameters such as threshold and key ratio changes for bloom filters at different sizes.

4.1 False Positive Rates

The false positive rates for each bloom filter are evaluated across a range of sizes, from 10000 kilobytes to 500000 kilobytes in intervals of 50 kilobytes. The FPR value is determined from querying each of the bloom filters on the entirety of the URLs categorized as malicious, where a mis-classification is if a query returns benign instead of malicious. The bloom filters evaluated during testing use the optimal hyperparameters for that size, which is determined during the training phase. Since the scores for each of the URLs were already generated by the classifier, the classifier prediction function is not actively invoked during training.

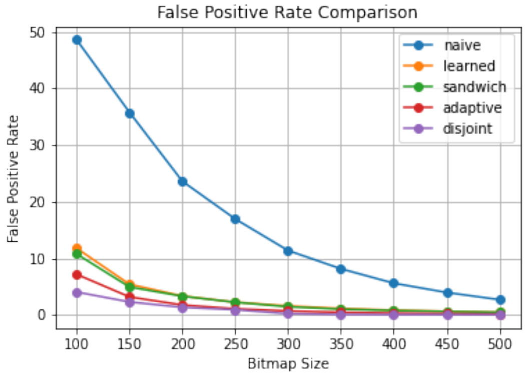


Figure 4: FPR Comparisons

Figure 4 displays the FPR results graphed as trend lines, with each marker representing an actual result. The y-axis denotes the percentage of malicious URLs that were mistakenly classified as positive, hence the "false positive". As a reminder, this percentage is out of the URL dataset's 400K total URLs that are marked as malicious. For the same amount of memory allocated, the learned bloom filters generally outperform the naive bloom filter by five to six times. Generally, the learned bloom filters achieve less than 10% FPR with just 250 kilobytes of storage. To get to less than 1% FPR, the sandwiched and learned bloom filters require about 350 kilobytes, while the adaptive and disjoint adaptive bloom filters hit this mark by 250 kilobytes.

Table 1: Learned BF Ideal Thresholds, Query Dist.

	150K	250K	350K	450K
Thresholds	0.7	0.8	0.85	0.9

The gap between the performance of the naive bloom filters and the learned variants can mainly be attributed to the accuracy of the classifier along with the increased storage space of the backup bloom filters relative to the number of examples. Compared to the naive bloom filter that hashes and stores the indices for all benign queries, a learned bloom filter will process 43% of the dataset when the threshold is set to 0.7. The below table 1 demonstrates how the ideal threshold for the learned bloom filter varies for different amounts of allocated memory. At smaller sizes where the backup bloom filter is more constrained, the classifier handles more of the queries. As the size increases, the backup bloom filter's accuracy improves. Therefore, the threshold increases, as more queries are forwarded to a larger bloom filter that is able to determine existence more accurately.

4.2 Learned Filters

Figure 5 presents an enlarged view of the FPR rates at larger bloom sizes. The overall trend lines from figure 4 along with this zoomed in look provide more insights on how the learned bloom filters compare. From comparing both the trajectories and the absolute false positive counts, it is possible to discern how commonalities and disparities in designs lead to relative improvements in performance.

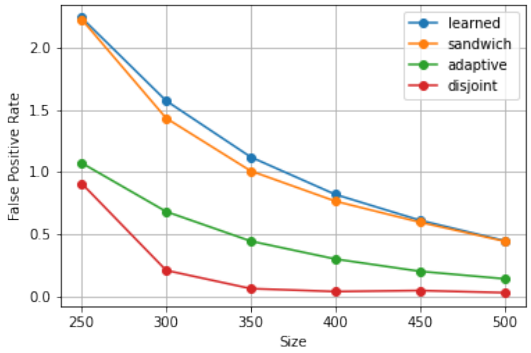


Figure 5: FPR Comparisons Enlarged

The learned and sandwiched bloom filters trend lines have a similar trajectory, with the sandwiched bloom filter's FPR trajectory being a consistent ratio smaller than the learned bloom filter, as we can see from the ratios in table 2. The sandwiched bloom filter consistently reduces the false positive rate by 10%. Inspecting the false positive URLs produced by the learned and sandwiched bloom filters reveals that the URLs misclassified by the sandwiched bloom filters are consistently and strictly a subset of the the URLs misclassified by the learned bloom filter.

Table 2: Learned & Sandwiched FPR Totals

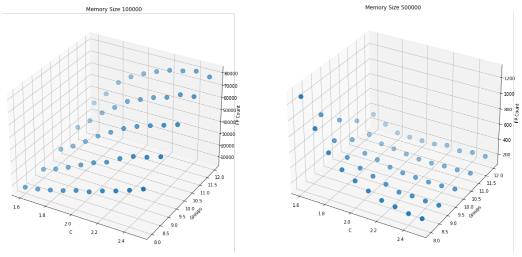
	150K	250K	350K	450K
Learned	18606	7719	3862	2111
Sandwiched	16974	7097	3476	1914
Ratio	1.0961	1.0876	1.111	1.1028

The difference between these sets — the malicious URLs that were false positives for the learned bloom filter, but not misclassified by the sandwiched design — were caught by the initial bloom filter for filtering out malicious negative URLs before sending the remainder to the classifier - this accounts for the improved performance of the sandwiched model. The consistent 10% reduction is a result of how the naive bloom filter's size scales with the amount of data being processed. This phenomenon is further affirmed by how storage is distributed between the initial and backup bloom filters in the sandwiched layout. Assuming a fixed threshold, all of the storage is allocated for the backup bloom filter when not much is provided (i.e. 100K - 250K). However, as the bloom filter size increases from 200K to 500K, the backup bloom filter becomes large enough to achieve an optimal, non-zero false positive rate. Once this threshold is reached, the extra storage does not lead to a reduction in false positives — at this point, allocating more bits to the initial bloom filter that can filter out some of the otherwise "irreducible" false positives is much more effective at increasing accuracy. These observations are consistent with the formal proof found in Mitzenmacher's paper [2].

4.3 Adaptive Filters

Another significant gap in performance can be found between the learned/sandwiched filters and the adaptive filters. The increased flexibility afforded by the adaptive filters' designs is empirically evident when looking at not only the FPR trend lines but also how the optimal values for the key ratio c and number of groups K hyperparameters change as the size of the bloom filter increases. Figure 5 provides insight into how these parameters affect the false positive rate for adaptive bloom filters with different sizes. While it's clear from the general FPR trend lines that Ada-BF works quite well, the ideal parameters at each memory size are quite different.

One of the first observations we can make is that as the memory size increases, the false positive rate decreases. Within the context of each memory size, there are more interesting relations between C and number of groups that varies with the memory size. First, smaller Ada-BF's seem to prefer fewer groups. For the bloom filter with 100K size, we can see that the false positive count steadily increases as the number of groups goes up for all values of C. On the other hand, as the size of the bloom filter in-



(a) 100K Size (b) 500K Size

Figure 6: FPR as a factor of k (number of groups) and c (ratio of keys) for adaptive bloom filters

creases, this trend still holds true for a larger C value, and therefore a larger ratio of keys, but is either not present and eventually reversed for Bloom Filters that are larger at smaller levels of C. The reasoning behind this can likely be attributed to overfitting. When the adaptive bloom filter is small, having too many groups may separate the distribution into too many segments, and that degree of tuning and specificity may lead to overfitting that exacerbates the false positive rate.

As the memory size increases, the shape of this scatter plot becomes more interesting, especially for the memory sizes of 250K - 350K. The optimal numbers seem to fall along the line of either 1. small C + large number of groups or 2. large C + fewer number of groups. I think this balance has to do with how many keys and separate into each group, and then what the ratio of the keys are. In this situation, it is a balance, rather than the extremities of the distribution, that would ensure not one single bucket of the learned distribution has too many keys. The corners of small C + small of groups or large C + large of groups can often lead to the learned index creating buckets that are uneven in size and tend to over-study some groups while not learning others enough.

Last but not least, another observation is that the c (ratio of keys) variable tends to have a much more significant effect on Ada-BF when it has a larger memory size, while number of buckets has a much greater influence on false positive counts for larger adaptive bloom filter.In general, the ratio of keys + number of groups values must be adjusted for Bloom Filters at different sizes in order to place the correct number of keys in each bucket. I think Ada-BF tends to lose its advantages when these numbers are not properly adjusted, in which case it begins to reduce to the performance of the learned Bloom Filter (very few groups + high disparity in ratio of keys across groups). In addition, for very simple distributions that do not inherently have that many buckets or don't have that many keys, a learned index might be overkill. However, these experiments do seem to reinforce the idea that the adaptive and disjoint bloom filters are much more accurate than standard Bloom Filters under the same memory budget, and learned indices would likely fare much better with richer datasets.

Acknowledgement

Thank you to the COS 513 course staff of Professor Adji Bousso Dieng, Dan Friedman, and Sunny Cui for feedback on my project throughout the duration of the Spring 2022 semester.

References

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

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

[3] Kraska, T., Beutel, A., Chi, E. H., Dean, J., and Polyzotis, N. *The Case for Learned Index Structures*. 2017. DOI: 10.48550/ARXIV.1712.01208. URL: https://arxiv.org/abs/1712.01208.

[4] Bonomi, F., Mitzenmacher, M., Panigrahy, R., Singh, S., and Varghese, G. "An Improved Construction for Counting Bloom Filters". In: 2006, 684–695. DOI: 10.1007/11841036_61. URL: https://doi.org/10.1007/11841036_61.

[5] Kirsch, A. and Mitzenmacher, M. "Less Hashing, Same Performance: Building a Better Bloom Filter". In: 33.2 (2008), 187–218. ISSN: 1042-9832.

[6] Dai, Z. and Shrivastava, A. *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.