Partitioned Learned Bloom Filter
6964 Final project

Abhishek Rajgaria, Ashish Tiwari

April 2024

Abstract

Bloom filters are space efficient data structures that are used to test whether an element is a member of a set or not with some probability. The trade-off with space is that it may result in some False positive outcomes, but it ensures that there are no False Negative outcomes. With the advent of applied Machine Learning in the data management domain, many researchers have introduced Learned models which could enhance the performance of Data management, Data query and other operations. One such researchers group Introduced a variation referred to as Learned Bloom Filters, in which a Machine Learning model is trained to predict a score and a backup bloom filter is used for avoiding any false negative samples, this improved performance in terms of the rate of false positives compared to memory used. These Learned Bloom Filters do not take full advantage of Learned models. To utilize the Learned model Partitioned Learned Bloom Filters were introduced.

1 Introduction

Bloom filters [1] are probabilistic data structures used for efficient set membership testing. They are space-efficient and support fast insertion and querying operations, with the trade-off of allowing a configurable false positive rate. Bloom filters have found applications in various domains, such as network routing, data de-duplication, and database indexing.

Depending on the domain, the dataset set could contain some structure which could be exploited by a machine learning model, which could act as a oracle to provide the answer to the membership request. This as the motivation researchers are applying machine learning models to exploit the data structure and get some prediction scores which could help in the inference or any other task at hand.

The main idea behind the Learned Bloom Filters (LBF) is that given a query input the likelihood that the input is in the set S can be deduced by some observable features which can be captured by a machine learning model. If we again take the example of URLs, we can train a Machine Learning model which can learn patterns for malicious URLs and can be utilized alongside the Bloom Filters, this approach was introduced by [2]. In the Learned Bloom Filter, the learned model is not exploited to the fullest, and the authers Vaidya [3] attempts to further be exploit the output of learned model by dividing the score in regions. So in Partitioned Learned Bloom Filtering, based on the output score of the learned model PLBF first identifies the region and then input is fed to the Bloom Filter of that particular region. We will go over the PLBF in detail in the design section.

Our contribution. The primary objective of this project is to develop and evaluate a partitioned learned bloom filter for efficient set membership testing, with a focus on URL classification and malicious URL detection. The project contributes the following:

- A character-level recurrent neural network (RNN) model for learning the distribution of URLs.
- An implementation of the partitioned learned Bloom filter and learned bloom filter.
- Evaluation and comparison of the vanilla Bloom filter, learned Bloom filter, and partitioned learned Bloom filter approaches in terms of false positive rates and memory usage in Mb.

2 Background

2.1 Traditional Bloom Filters

A Bloom filter is a probabilistic data structure that represents a set of elements using a bit vector and a set of hash functions. Elements are inserted into the filter by setting the corresponding bits in the bit vector based on the hash function outputs. Membership queries are performed by checking if all the corresponding bits are set in the bit vector. Bloom filters guarantee a zero false negative rate (elements in the set will never be reported as not present), but they allow for a configurable false positive rate (elements not in the set may be reported as present). While Bloom filters are efficient in terms of space and time complexity, they have some limitations:

- **Equal Treatment of Elements:** Traditional Bloom filters treat all elements equally, regardless of their importance or frequency in the dataset. This can lead to sub-optimal performance when the data distribution is skewed or when certain elements have higher relevance.
- **Static Configuration:** The false positive rate and memory usage of a bloom filter are determined at initialization and cannot be adapted to changes in the data distribution or application requirements.

2.2 Learned Bloom Filters

Learned Bloom filters aim to address the limitations of traditional Bloom filters by leveraging machine learning models to learn the underlying data distribution and optimize the filter's configuration accordingly. Instead of using fixed hash functions, learned Bloom filters use a machine learning model to map elements (e.g., URLs) to a continuous output space (e.g., $[0, 1]$). A threshold value is then chosen to classify elements as members or non-members of the set. By learning the data distribution and optimizing the threshold value, learned Bloom filters can achieve better trade-offs between false positive rates and memory usage compared to traditional bloom filters.

One major thing that needs to be addressed in the Learned Bloom Filter is identifying the threshold value. We follow the steps provided in [2]. Let's denote the False Positive Rate (FPR) of our model to be FPR_{model} which is evaluated as $\frac{FP}{FP+TN}$. Since this learned model is not represented by the universe, so there could be elements which may be identified as a false negative from the model, but in order to preserve the 0 FN count of Bloom Filter, we will add a backup bloom filter, in which we will add those element which are classified as Negative but are part of the set i.e. False Negative during the training. The FPR_{BBF} of this backup bloom filter and the FPR_{Model} of the model should be $FPR_{Model} = FPR_{BBF} = \frac{TargetFPR}{2}$. The threshold value for which this condition is satisfied is a desirable threshold value. Infact, the FPRs could less than or equal to.

The key steps in the Learned Bloom Filter are:

1. Train a Machine Learning model for the dataset. (One key point, the Machine Learning Model shouldn't be too large, so using large Machine Learning models may not solve our problem).
2. Find the optimal threshold value which satisfies the above constraints.
3. Handle false negatives (elements incorrectly classified as non-members) using Backup Bloom filter.

2.3 Partioned Learned Bloom Filter

Partitioned learned Bloom filters extend the idea of learned Bloom filters by partitioning the data into multiple regions based on the learned model scores. Separate Bloom filters are used for each region, with optimized false positive rates tailored to the data distribution in that region. This approach provides more fine-grained control over the false positive rates and memory usage, in theory it has potential to achieve better performance as compared to a single learned Bloom Filter, but we will see in our case, our neural network model does a pretty good job in separating the scores of the malicious and benign URLs.

The key steps in the PLBF algorithm are:

1. Train a Machine Learning model for the dataset. (One key point, the Machine Learning Model shouldn't be too large, so using large Machine Learning models may not solve our problem).
2. Partition the data into regions based on the learned model scores and a set of threshold values. The set of Threshold values are needed to be evaluated by maximizing the KL divergence between key and non-key scores.
3. Calculate the optimal false positive rates for each region using an iterative algorithm that minimizes the overall false positive rate while satisfying the target false positive rate constraint.
4. Create separate Bloom filters for each region, with the corresponding optimal false positive rates.

The authors frame the space vs false positive rate trade off as on optimization problem with the aim of - Minimizing the total size of the backup bloom filters while maintaining the overall false positive rate comparable or lower to target false positive rate desired by the user.

$$\min_{f_i=1\ldots k,t_i=0\ldots k} \sum_{i=1}^{k} \text{Size of } i\text{th backup Bloom filter}$$

Constraints - Overall False Positive Rate $\leq$ Target False Positive Rate.

On high level we could say that, space saved by the Partitioned Learned Bloom Filter compared to traditional Bloom Filter is proportional to KL Divergence of key and non-key score distribution. PLBF Space saved $\propto$ KL_Div (key scores, non-key scores)

$$\text{fpr of } i^{th} \text{ region} \propto \frac{\text{Proportion of keys}}{Proportion of Non-keys}$$

The PLBF algorithm aims to achieve a lower overall false positive rate and better memory usage compared to traditional Bloom filters and learned Bloom filters by exploiting the varying data distributions across different regions.

2.4 Gated Recurrent Unit

The Gated recurrent unit (GRU) is a type of recurrent neural network (RNN) that is mainly useful for learning sequential data. It is an improvement over the vanilla RNNs which faces vanishing gradient and exploding gradient problem. It is very much similar to a very famous RNN i.e. LSTM (Long Short Term Memory) recurrent neural network which has separate memory/forget and current input gate. In GRU, the memory/forget and input gate is combined to a single update gate. This simplification reduces the number of parameters in the model and makes it computationally less expensive while still providing comparable performance.

3 Motivation

Traditional Bloom filters treat all elements equally, leading to sub-optimal performance when the data distribution is skewed or when the elements have varying importance or relevance. Learned Bloom filters aim to address this limitation by leveraging machine learning models to learn the underlying data distribution and optimize the filter's configuration accordingly. Partitioned learned Bloom filters (PLBF) [3] extend this idea by partitioning the data into multiple regions and using separate Bloom filters for each region, allowing for more fine-grained control over the false positive rates and memory usage.

Personal motivation to pursue this project was, it allowed us to work in the domain of data and how machine learning/ current deep learning models could help us utilize to learn from the data and increase the efficiency and effectiveness the task available at hand.

4 Data used

For our project, we utilized the URLs dataset, comprising a total of $400,000$ data points. Among these, 23% were identified as malicious URLs, while 77% were classified as benign URLs. The dataset was obtained from Kaggle [4]. Initially, we had also planned to utilize the EMBER dataset for our study. However, due to time constraints, we were unable to complete the acquisition and preprocessing of this dataset.

4.1 Data Preprocessing

The dataset was preprocessed for training our model which involved following steps:

- Two datasets were loaded:
  - `urldata.csv` containing URLs with their labels.
  - `malicious_phish.csv` containing malicious URLs with their labels.
- Mapped 'type' to binary labels, selected 'url' and 'result'. And concatenated the datasets vertically.
- Removed duplicate entries from dataset if present.
- Split the combined dataset into training, validation, and test sets. (this is ideal, but we have only split our dataset into train and test)

5 Design

We have implemented 3 versions of the Bloom Filters and in this section we will be going over them and how we have implemented it.

5.1 Vanilla Bloom Filter

It is a simple bloom filter, which we use straight from the python Bloom Filter 2 library. We created class over the python library to accommodate our requirements and make our code modular. It takes two parameters maximum element that could be added to the bloom filter and error rate, which is nothing but the target false positive rate. For getting the size of this filter we are dividing the number of bits used by the filter with $(1024 * 1024)$, which will give us the output in Mb.

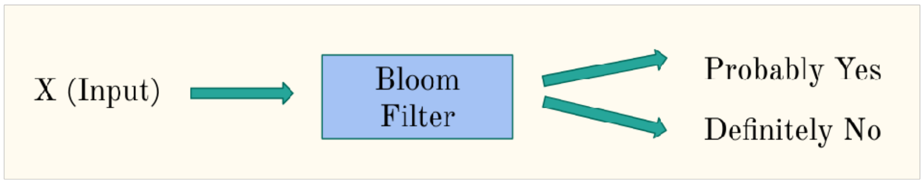

Figure 1: Architecture of Vanilla Bloom Filter

5.2 Learned Bloom Filter

In this section, we will going over the steps mentioned in section 2.2 in details. Learned Bloom Filter uses a Gated Recurrent Unit Neural network for learning the structure from the URL dataset. We used GRU, because URL is sequential text data and GRU works good on textual data, plus it has less number of parameters which will keep our model size small. We have Binary Cross entropy loss, as our learning task is a binary classification task along with the Adam optimizer for the learning rate 0.001. The final layer is Sigmoid layer, which gives output in the range 0 to 1.

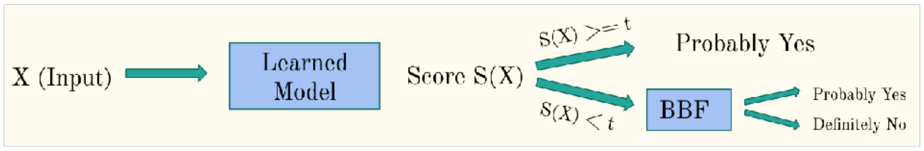

Figure 2: Architecture of Learned Bloom Filter

Now, after training the model, we need to find a threshold value which satisfy the constraints need to keep overall false positive rate of the learned bloom filter comparable to the target false positive rate. We have applied binary search to get the optimal threshold value. For each threshold value, we compute the false positive rate and if it is greater than the half of the target FPR, then we increase the floor or our search space otherwise, we have the threshold value keeps the model FPR to be less than or equal to the target FPR.

Once, you have the optimal threshold value, we need to find the false negative URls which are malicious URLs that are predicted as benign or negative by our model. We will add these URLs in the Backup bloom filter (here we use our Vanilla Bloom Filter) with error rate of half of the target false positive rate.

For getting the size of this filter we are dividing the number of bits used by the filter with $(1024 * 1024)$, which will give us the output in Mb, Plus the size of the learned model in Mb.

5.3 Partitioned Learned Bloom Filter

For training the model, we followed the same steps as mentioned in the Learned Bloom Filter(LBF) in section 5.2

Unlike LBF, in partitioned learned bloom filter, the score range needs to be divided into partitions and we have used 5 partitions. (Authors recommend to use 5-10 partitions). We tried with higher partition values, but then most of the higher partitions will get FPR value of 1 (we will discuss what it means to have FPR value of 1 for a region.)

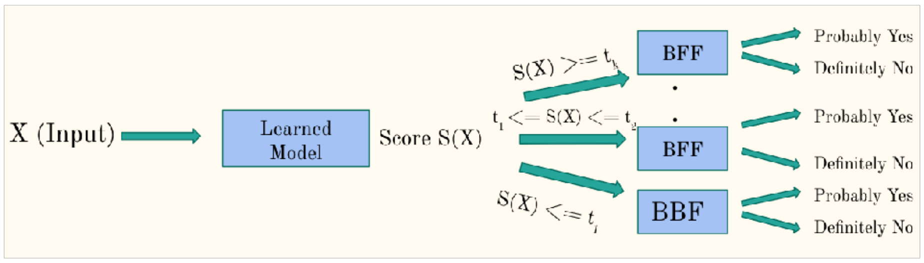

Figure 3: Architecture of Partitioned Learned Bloom Filter

The idea with partitioned learned bloom filter (PLBF) is to partition the score space into multiple regions using various threshold values. Each region employs a separate backup bloom filters, with the flexibility to choose different target false positive rates for each region. For each viable region we have created vanilla bloom filter with the positive rate provided by the PLBF algorithm.

Each sample belongs to a specific region, as the model outputs in the range of 0 to 1. Therefore, each sample is added to some Bloom filter (this is ideal scenario). As we hinted that few regions could have False positive rates of 1 and bloom filter can't have a false positive rate of one. So we improvised and if the region has a false positive rate of 1, then this region do not require Bloom Filter, as the distribution suggest that any sample which have score in this region is definitely a malicious URL. This improvisation save us a lot of memory because if we had added all the malicious URL, then the memory usage could be very comparable to the vanilla bloom filter, plus the learned model.

For getting the size of this filter we are dividing the number of bits used by the filter with (10241024) and summing for all region's bloom filters, which will give us the output in Mb, Plus the size of the learned model in Mb.

6 Evaluation

To perform evaluation, we have chosen three target false positive rates (0.005, 0.01, 0.05). This is the error rate of the bloom filters, since the usecase of these filters is to make the membership request faster and consume less memory. We have compared the partitioned learned bloom filter on the URL dataset with the original bloom filter (we have referred to this as vanilla bloom filter) and learned bloom filter. In comparison, we have created a table, which list the target false positive rate and memory used by the particular bloom filter along with learned model size (if used).

Since URL could be of arbitrary length, but the GRU (our machine learning model) needs the input of specific size, therefore we have added limited the size of URL to be of 32. (we used this because, average URL length was around 25). If the URL length was greater than this we truncated it, otherwise added padding.

Table 1: Comparison of False Positive Rates and Memory Usage for Different Bloom Filter Types, The Learned Bloom Filter and Partitioned Learned Bloom Filter Memory usage also include size of the Machine Learning Model used (GRU) which is of 56KB = 0.448 Mb

.

Filter Type	**False Positive Rate (%)**	**Memory Usage (Mb)**
Vanilla Bloom Filter	0.005	2.32
	0.01	2.02
	0.05	1.31
Learned Bloom Filter	0.005	0.462
	0.01	0.460
	0.05	0.456
Partitioned Learned BF	0.005	0.658
	0.01	0.596
	0.05	0.459

6.1 Inference

From the table 1 we can clearly see that memory usage has been drastically reduced for Learned Bloom Filter and the Partitioned Learned Bloom Filter as compared to the Vanilla Bloom Filter. The main reason for this, is that in the Vanilla Bloom Filter all the malicious URL needs to be added, to accommodate such large number of URLs while keeping a low false positive rate, the vanilla bloom filter requires a very big size of bits array. While in the case of the LBF and PLBF, most of the samples are handled by the model and are not needed to add in the backup bloom filter or region bloom filter which saved a lot of space. The training time increases with vanilla bloom filter, LBF and PLBF in sequence, it could be attributed to the complexity of the filters.

Also, with the increase in the False positive rate, the amount of memory used keeps decreasing and which also make sense, as the error rate increases we require less number of bits in the binary bits array as we are okay with this much error, So based on the domain and task, the desirable error rate could be used to have lower memory usage.

One interesting observation, the memory usage for the Learned Bloom Filter is more than that of the Partitioned Learned Bloom Filter, but as we go towards the higher error rate the difference keeps decreasing. Now a question comes, shouldn't the memory of LBF be lower than the PLBF? because we are utilizing the learned model output space more robustly. But it turns out that, if the model does a good job in differentiating the data in case of binary classification, then we do not require large number of partitions.

7 Conclusion

So, overall the performance of the Bloom filter with machine learning model increases and memory usage decreases. The Partitioned Learned Bloom filters should be used when there are multiple classes or the output of the model is distributed over the score range. Bloom filters have one disadvantage that we cannot delete URLs from them once added, but from the learned model, we could do something similar to deleting the URL, that applying forget machine learning algorithm on the learned model.

References

[1] Y. Lu, B. Prabhakar, and F. Bonomi, "Bloom filters: Design innovations and novel applications," January 2005.

[2] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 489–504. [Online]. Available: https://doi.org/10.1145/3183713.3196909

[3] K. Vaidya, E. Knorr, T. Kraska, and M. Mitzenmacher, "Partitioned learned bloom filter," *CoRR*, vol. abs/2006.03176, 2020. [Online]. Available: https://arxiv.org/abs/2006.03176

[4] S. Sadiq, "Benign and malicious urls," https://www.kaggle.com/datasets/samahsadiq/benign-and-malicious-urls, 2022, kaggle.