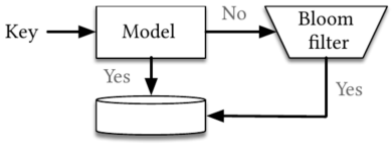
Exploring Machine Learning Enhanced Bloom Filters

Minh Nguyen (mn78@rice.edu), Utkarsh Bahuguna (ub5@rice.edu)

Abstract

Bloom filters are extremely space efficient data structures for membership verification, but since they are probabilistic, traditional Bloom filters frequently have a fixed false positive rate and are not flexible enough to adjust to changing data patterns. Also in some applications, the size of the membership set might be so large that even the bloom filter might exceed space limitations. In this project, we will investigate how machine learning can improve Bloom filters in such situations. In some other cases, more information beyond just membership verification is required while still being space efficient. Given that machine learning (ML) models are able to adjust and learn from the dataset's features, bloom filters enhanced with ML can potentially overcome some such shortcomings of traditional Bloom filters.

In this project, we implement the following variations of Bloom Filters enhanced by utilizing Machine Learning:



- Pre-filter Learned Bloom Filter
    - Where a learned ML model is used as a pre-filter for the Bloom Filter
- Anomaly Detection Bloom Filter
    - Where an ML model is utilized after the bloom filter as an additional layer of verification to reduce impact of false positives
- Class-aware Bloom Filter
    - Where an ML model is used after the bloom filter to give additional information about the class of the item, rather than just its presence in the filter.
- Class-wise Bloom Filters
    - Where a multi-class ML classifier is used to direct queries to the relevant bloom filter, possibly reducing FP rates in each class-specific filter.

Literature Survey

There has been a surge in interest in the use of machine learning to Bloom filters, with multiple research putting forth creative methods to improve its conventional capabilities. In a paper, Mitzenmacher (2018) proposes the idea of "Model-Based Bloom Filters," in which the parameters of the filter are optimized by a machine-learning model that forecasts the probability that an element will be included. The results of this study demonstrate that it is possible to reduce the false positive rate by combining predictive models with Bloom filters without significantly sacrificing computing performance.

Kraska et al. (2018) additional groundbreaking paper presents the concept of learnt index structures. While their work primarily focuses on database indexing, the idea behind replacing conventional algorithms with learning models directly affects how Bloom filters are improved. Their discovery opens the path for data structures that change and adapt with the data they hold,

resulting in more effective querying and storage, by teaching models to predict items' locations or existence within a dataset.

These studies provide the framework for our project. By utilizing machine learning's capacity to learn from data and identify relevant features, we seek to explore the issues of lowering the false positive rate in Bloom filters, reducing the memory footprint, reducing the rate of false negatives, and providing additional information beyond simple membership verification.

Hypothesis

We hypothesize that machine learning models can effectively complement Bloom filters and improve performance metrics like space utilization, false positive rates, etc. without compromising on the inherent benefits like speed of membership checks.

Experimental Setup

Datasets

We will primarily use two datasets for this project:

- Malicious URLs: an extensive collection of 651,191 URLs, making it highly relevant for testing the efficacy of Bloom filters in web security applications. It encompasses a broad spectrum of URL types, categorized into benign (safe) URLs and various types of malicious URLs such as defacement, phishing, and malware.
- NSL-KDD: a widely used benchmark for evaluating network intrusion detection systems, featuring a diverse range of simulated attack scenarios alongside normal network traffic.

Methodology

We first train binary classifiers and multi-class classifiers for both the datasets. The binary classifier for the malicious URLs classifies a URL as either benign or malicious, and that for the NSL-KDD dataset classifies an entry as either normal or attack. The multi-class classifier for malicious URLs categorized into four labels: benign, defacement, phishing, and malware. For NSL-KDD, the classes are normal, denial-of-service, probe, and other.

The goal is to train ML models which are lightweight, reasonably accurate, and have low inference times. We experiment with two different model types: 1D CNN models for the binary classifiers, and XGBoost for the multi-class classifiers. While CNN models edge out in test accuracy, XGBoost models are much smaller and faster in comparison. The difference can be seen between the binary and multi-class variants. Model performances on the test sets can be seen in the classification reports given in figures 1-4. We also include some of the more complex model architectures and training parameters in the appendix.

	precision	recall	f1-score
Benign	0.99	0.99	0.99
Malicious	0.98	0.97	0.98
accuracy			0.98
macro avg	0.98	0.98	0.98
weighted avg	0.98	0.98	0.98

Figure 1: Malicious URLs: CNN binary classifier

	precision	recall	f1-score
Normal	0.98	0.99	0.99
Attack	0.99	0.98	0.99
accuracy			0.99
macro avg	0.99	0.99	0.99
weighted avg	0.99	0.99	0.99

Figure 2: NSL-KDD: CNN + LSTM classifier

	precision	recall	f1-score
benign	0.97	0.99	0.98
defacement	0.95	0.98	0.97
malware	0.97	0.87	0.92
phishing	0.90	0.81	0.85
accuracy			0.96
macro avg	0.95	0.91	0.93
weighted avg	0.95	0.96	0.95

Figure 3: Malicious URLs: XGBoost multi-class classifier

	precision	recall	f1-score
DOS	1.00	1.00	1.00
Other	0.98	0.81	0.89
Probe	0.99	0.97	0.98
normal	0.99	1.00	0.99
accuracy			0.99
macro avg	0.99	0.94	0.96
weighted avg	0.99	0.99	0.99

Figure 4: NSL-KDD: XGBoost multi-class classifier

With the trained models at hand, the following is the set up for the four variants:

1. **Pre-filter Learned Bloom Filter**: The ML model functions as the prior checkpoint to the traditional Bloom filter. The model produces a predictive score $f(x)$ for each element x, indicating the likelihood that x is a member of the set. The threshold tau is a predefined value that determines whether an element is considered likely to be in the set ($f(x) \geq tau$) or not ($f(x) < tau$). If $f(x) \geq tau$, the element is deemed likely to be a member and is passed on to the Bloom filter for verification.
   Otherwise, the element is presumed not to be a member, and the Bloom filter is bypassed, which can potentially avoid an unnecessary query and hence a false positive. A tau a value of 0.9 is appropriate, as it sets a high threshold for classification certainty, aligning with the model's demonstrated accuracy and reducing the chance of false positives. Additionally, the elements can be filtered while being added to the bloom filter itself, which will reduce the overall size significantly, but this is something we do not explore here.

2. **Anomaly Detection Bloom Filter**: The ML model is employed after the Bloom filter. Here, the Bloom filter operates first, and every element x that it claims to be present in the set is then verified by the ML model. If the Bloom filter indicates that x is present and $f(x)$ is less than tau, the ML model contradicts the Bloom filter's claim, suggesting that x might be a false positive. Conversely, if $f(x)$ is greater than or equal to tau, the ML model confirms the Bloom filter's claim, reinforcing the belief that x is indeed a member of the set. This subsequent verification can reduce the FPR by catching and reevaluating potential false positives indicated by the Bloom filter.

3. **Class-Aware Bloom Filter**: Like the anomaly detection bloom filter, the ML model is used after the Bloom filter, but instead of giving a binary decision, the ML model predicts the class of the item, and its confidence. This additional information can then be used in downstream systems to appropriately handle different cases. For this variant, we do not use a threshold in order to also get a comparison with variant 2.

4. **Class-Wise Bloom Filters:** Instead of using a single bloom filter, multiple bloom filters – one for each class – are used and a multi-class classification model directs queries to the appropriate bloom filter. As with the first variant, here too if the model determines the item to not be a member, query to any bloom filter is avoided. Having separate bloom filters by class can also reduce the false positive rates within a class especially when the class distribution is skewed.

For each of the variants we proceed as follows:

Take 50% of the membership eligible items (malicious URLs or attacks) and insert them into the bloom filters (single or class-wise). This simulates the lack of complete information in the real world, where one can never have all the malicious URLs or attacks pre-populated. For baseline comparison, a bloom filter is used without any ML model and the same items are populated. All bloom filters use a theoretical FP rate of 0.001. For testing, 10% of the original dataset items are taken and the following metrics observed:

Metrics

- False positive rate (FPR): Measures the frequency at which the set up incorrectly identifies an element as being in the set when it is not.
- Missed rate/False negative rate: Different from the false negative rate for bloom filters (which is always 0), this rate measures how many malicious URLs or attacks were not flagged as such by the setup. Note that in our baseline set up this will not be 0, because we did not populate the bloom filter with all the eligible items.
- Average Query time: The amount of time taken by the set up to give a result for an item.
- Memory footprint: The total memory size taken by the set up.

Results

The following results are observed for the four variants. Note that not all metrics apply equally for all the variants, since each variant tries to optimize some specific metric and may trade off on the others.

```
Comparison of False Positive rates
  FP Rate (Normal BF)  FP Rate (ML + BF)  FP Rate (BF + ML)
            0.000353                0.0           0.000307

Comparison of Missed Rates
  Missed Rate (Normal BF)  Missed Rate (ML + BF)  Missed Rate (BF + ML)
                 0.162671               0.012347               0.012347

Overall Performance Metrics:
                              Normal BF     ML + BF     BF + ML
Average Query Time (seconds)   0.000003    0.000004    0.000004
Memory Footprint (kB)        216.406250  611.148438  611.148438
```

Figure 5: Binary Classification [variants 1 (ML+BF) and 2 (BF+ML)] - Malicious URLs

```
 Comparison of False Positive rates
   FP Rate (Normal BF)  FP Rate (ML + BF)  FP Rate (BF + ML)
             0.001667                0.0           0.001111

 Comparison of Missed Rates
   Missed Rate  Missed Rate (ML + BF)  Missed Rate (BF + ML)
       0.46519               0.025323               0.025323

 Overall Performance Metrics:
                               Normal BF      ML + BF      BF + ML
 Average Query Time (seconds)   0.006902     0.006849     0.006948
 Memory Footprint (kB)         52.445312  1311.500000  1311.500000
```

Figure 6: Binary Classification [variants 1 (ML+BF) and 2 (BF+ML)] - NSL-KDD

```
Performance Comparison:
    Approach   FP Rate  Missed Rate  Avg Query Time (seconds)  Memory (kB)
0      BF+ML  0.009858     0.011355                  0.008017   196.906250
1  Normal BF  0.000593     0.165148                  0.000043   187.953125
```

Figure 7: Multi-class Classification [variant 3] - Malicious URLs

```
Performance Comparison:
    Approach   FP Rate  Missed Rate  Avg Query Time (seconds)  Memory (kB)
0      BF+ML  0.001350     0.001985                  0.023887    54.906250
1  Normal BF  0.000873     0.228308                  0.007382    52.445312
```

Figure 8: Multi-class Classification [variant 3] - NSL-KDD

```
Comparison of FP and FN Rates:
     Class  FP Rate (ML+BF)  FP Rate (Normal BF)  FN Rate (ML+BF)
  phishing         0.000037             0.000110         0.162193
    benign         0.000141             0.001733         0.000000
defacement         0.000073             0.000092         0.013988
   malware         0.000000             0.000049         0.079880

Overall Performance Metrics:
                                  ML+BF   Normal BF
Average Query Time (seconds)    0.010003    0.000038
Memory Footprint (kB)         198.945312  187.953125
```

Figure 9: Multi-class Classification [variant 4] - Malicious URLs

```
Comparison of FP and FN Rates:
 Class  FP Rate (ML+BF)  FP Rate (Normal BF)  FN Rate (ML+BF)
normal         0.000000             0.001881         0.000000
   DOS         0.286601             0.287719         0.002861
 Other         0.004968             0.005930         0.177966
 Probe         0.048808             0.050824         0.026160

Overall Performance Metrics:
                                  ML+BF  Normal BF
Average Query Time (seconds)   0.026785   0.007628
Memory Footprint (kB)         56.945312  52.445312
```

Figure 10: Multi-class Classification [variant 4] - NSL-KDD

The following common observations can be made from the results above:

1. The FP rates are always lower whenever the ML model is used with the bloom filter along with a prediction confidence threshold is used.
2. The missed rates are much lower whenever ML models are used in conjunction with the bloom filter. In real-world scenarios, this would be a huge win since keeping all eligible items in a bloom filter at all times would never be possible.
3. The memory footprint is higher for ML variants (only slightly for XGBoost, much more for CNN models), considering the size of the ML model. As we previously noted, this is not a fair representation of the memory requirements for the ML variants though, since when using ML variants, the elements with which the bloom filter needs to be populated to get similar results will be much less than that for a standalone bloom filter.
4. The average query times are much higher with the ML variants (note that figures 5 and 6 show erroneous calculations which do not include the model inference time in the query time).

Conclusion

In line with our hypothesis, we have shown how different variations of combining ML models with bloom filters can achieve significant improvements in certain metrics (FP rate, missed rate,

memory consumption) and provide additional intelligence for downstream systems. Although, a clear tradeoff exists between accuracy and query times, enhancing bloom filters with ML models can provide valuable benefits in many applications especially with severe space constraints, or where the cost of a false positive or a missed negative is high. By incorporating ML models, we can adapt more dynamically to the data patterns, potentially creating a more intelligent system that can evolve with the data it processes.

References

Kraska, Tim, et al. “The Case for Learned Index Structures.” *arXiv.org*, arXiv, 30 Apr. 2018, arxiv.org/abs/1712.01208.

Bhattacharya, Arunava, et al. “New Wine in an Old Bottle.” *Proceedings of the VLDB Endowment*, doi.org/10.14778/3538598.3538613.

Mitzenmacher, Michael. “A Model for Learned Bloom Filters and Optimizing by Sandwiching.” *arXiv preprint*, arXiv, 2018, arxiv.org/abs/1802.00884.

Mohi-ud-din, Ghulam. “NSL-KDD.” *IEEE Dataport*, 2018, dx.doi.org/10.21227/425a-3e55.

Siddhartha, Manu. “Malicious URLs.” *Kaggle*, www.kaggle.com/datasets/sid321axn/malicious-urls-dataset.

Appendix

A 80-20 training-test split is used for all the models.

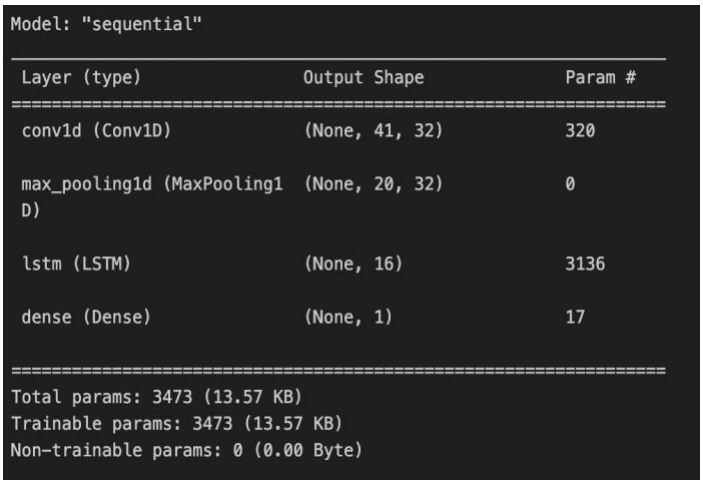
CNN Model – Malicious URLs

1. **Input and Tokenization**: The model takes URLs as input. Each URL is tokenized character-by-character using a custom tokenizer that maps each character in a predefined alphabet to a unique integer. The alphabet includes lowercase letters, numbers, and various symbols, making it comprehensive for URL analysis.
2. **Embedding Layer**: An encoded embedding layer is used to transform these integer sequences into dense vectors of fixed size (69 dimensions). This layer captures the character-level features of URLs.

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input (InputLayer)          [(None, 1014)]            0

 embedding (Embedding)       (None, 1014, 69)          4830

 conv1d (Conv1D)             (None, 1011, 64)          17728

 max_pooling1d (MaxPooling1  (None, 505, 64)           0
 D)

 flatten (Flatten)           (None, 32320)             0

 dense (Dense)               (None, 64)                2068544

 dropout (Dropout)           (None, 64)                0

 dense_1 (Dense)             (None, 4)                 260

 dropout_1 (Dropout)         (None, 4)                 0

 dense_2 (Dense)             (None, 1)                 5
```

3. **Convolutional Layers**: The model employs a Conv1D layer with 64 filters and a kernel size of 4. This configuration helps the model learn spatial hierarchies of features from the character sequences, which is crucial for recognizing patterns in URLs that might indicate malicious intent.
4. **Pooling and Flattening**: A MaxPooling1D layer follows, reducing the spatial dimensions and hence the computational complexity. The Flatten layer then converts the pooled feature maps into a single long vector, suitable for input into dense layers.
5. **Dense Layers and Dropout**: The model includes dense layers with 64 and 4 neurons, respectively, both employing the ReLU activation function. Dropout layers with a rate of 0.1 follow each dense layer to prevent overfitting by randomly setting input units to 0 at each update during training time.
6. **Output Layer**: A final dense layer with a sigmoid activation function produces a binary output, indicating the likelihood of the URL being malicious.
7. **Compilation and Training**: The model is compiled with the Adam optimizer and binary cross-entropy loss function, both standard choices for binary classification tasks. It is trained for 10 epochs with a batch size of 256.

CNN + LSTM Model – NSL-KDD



1. **Conv1D Layer**: It starts with a convolutional layer with 32 filters and a kernel size of 9. This is suitable for the NSL-KDD dataset as it allows the model to capture the spatial hierarchies in the input data by learning from the sequence of features in the network traffic data. The padding is set to "same" to keep the output size equal to the input size, ensuring no loss of edge information in the input.
2. **MaxPooling1D Layer**: The pooling layer with a pool size of 2 reduces the dimensionality of the data by down-sampling, which helps in reducing the computational cost and overfitting.
3. **LSTM Layer**: This layer captures long-term dependencies between time steps of sequence data, which is crucial for understanding the temporal dynamics of network traffic, making it highly relevant for the NSL-KDD dataset. It includes 16 units and a dropout rate of 0.2 to prevent overfitting.
4. **Dense Layer**: The model concludes with a dense layer with a single unit and a sigmoid activation function to perform the binary classification, indicating whether an input sample has an attack 1 or not 0.
5. **Training Parameters**: The model is trained with a batch size of 250 and for 10 epochs. These parameters are chosen to balance between the computational efficiency and the ability of the model to generalize. A validation split of 10% allows the model to be validated on unseen data during training.
6. **Loss Function and Optimizer**: The model uses binary cross-entropy as the loss function, which is standard for binary classification problems, and the Adam optimizer, known for its efficiency in stochastic gradient descent.