# A Mixed Analysis of Learned Bloom Filters

Kevin Rao

CS223 Final Paper

Bloom filters are a probabilistic data structure that determine membership in a key set and is well known for guaranteeing no false negatives. The Bloom filter can be thought of as a classifier based on membership of the set, so recent literature has explored how we can improve the performance of Bloom filters by using machine learning classifiers. Two such data structures are the Learned Bloom Filter and the Sandwiched Bloom Filter.

In this work, we first review the definition of the standard Bloom filter and some results regarding its false positive rate. We then describe the Learned Bloom Filter and the Sandwiched Bloom Filter and define their respective false positive rates. We also present bounds on the size of the learned functions to explore when a Learned Bloom Filter or Sandwiched Bloom Filter performs better than a standard Bloom filter with the same amount of memory. Finally, we present empirical data drawn from implementations of the data structures.

All of the classes and code that were used to generate the data in this paper are available at `https://github.com/kevinrao99/cs223-bloom`

## 1 Introduction

In "The Case for Learned Index Structures"[3], the authors suggest that using the power of machine learning, we can provably improve the performance of certain index structures, such as Bloom filters and B trees. "A Model for Learned Bloom Filters, and Optimizing by Sandwiching" [2] formalizes this idea with Bloom filters and present the Sandwiched Bloom Filter, which uses an initial filter and backup filter to support a learned function that attempts to classify queries.

In this paper we report on the findings in [2] and support them with empirical data generated by implementing the data structures in Python with a neural net as the learned function.

## 2 The Standard Bloom Filter

We begin our paper by reviewing the construction of the standard Bloom Filter, which was first described by Burton Howard Bloom in 1970. The Bloom filter is a probabilistic data structure that stores a set of keys and handles query requests. Whenever an element from the universe is queried, the Bloom filter must decide if the element is a key or not. The Bloom filter's most notable property is that it can guarantee that it never responds with a false negative, so the analysis of Bloom filters always revolves around the false positive rate.

### 2.1 Definition of Bloom Filter

A Bloom filter is used to represent a set $S$ of $n$ elements $\{s_1, s_2, \dots s_n\}$ from a universe $U$. It stores an array $M$ of $m$ bits and uses $k$ independent hash functions $h_1, h_2, \dots h_k$, where $h_i : U \to [0, 1, \dots m-1]$. We make the typical assumption here that the hash functions are perfect and independent, or given any element $u \in U$ and any value $i \in [0, 1, \dots m-1]$, for all hash functions $h_j$

$$\Pr[h_j(u) = i] = \frac{1}{m}$$

When inserting each element $s \in S$, the Bloom filter computes the values $h_1(s), h_2(s), \dots h_k(s)$, and sets the bit in $M$ at index $h_i(s)$ to true for all $1 \le i \le k$. Notably, the same index can be set true more than one time. When queried with an element $u \in U$, the Bloom filter returns

$$\bigwedge_{i=1}^{k} M[h_i(u)]$$

If $u$ is in our key set, the bits at $h_1(u), h_2(u), \dots h_k(u)$ have all been set to true when we inserted the contents of $S$ into our Bloom filter, so we are guaranteed to return a hit for all elements in $S$. This is precisely the no false negatives guarantee of the Bloom filter. If $u$ is not in $S$, then we can compute the probability that the bits at $h_1(u), h_2(u), \dots h_k(u)$ have all been set to true by chance, and our Bloom filter returns a false positive.

## 2.2 False Positive Rate

To find the false positive rate of our bloom filter, we take advantage of the fact that our $k$ hash functions are independent and perfect in the sense that they map every element in $U$ uniformly to the set $[0, 1, \dots m]$. Suppose we pick some array index $i$. Since we have a total of $n$ elements in our key set and we compute $k$ hash functions for each, we can model the problem as randomly inserting $kn$ balls uniformly into $m$ bins. The probability that any given ball is placed in our bin is $\frac{1}{m}$, so the probability that no ball has been placed in our bin is

$$\left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}$$

And by linearity of expectations the expected fractions of our $m$ bins that are empty after inserting our key set also $e^{-kn/m}$. Thus, the probability that an element $u \in U \backslash S$ is falsely identified as an element of our key set is the probability that all $k$ bins $h_1(u), \dots h_k(u)$ have at least one ball in them. This gives us the false positive rate

$$\Pr[u \text{ gives a false positive}] = (1 - e^{-kn/m})^k$$

It's worth noting that this false positive rate is actually an expected value, because it's based on the expected number of bins that are still empty after throwing $kn$ balls into bins. However, we can use the standard concentration bounds to show that with high probability the false positive rate is very close to its expectation. Thus, we usually talk about the false positive rate of a Bloom filter as a constant given the parameters $m, n, k$, even though it's technically a random variable.

Due to the nature of the false positive rate, the ratio $\frac{m}{n}$ is often kept constant. If we wish to solve for the optimal value of $k$, we can use the trick from [1] and observe that

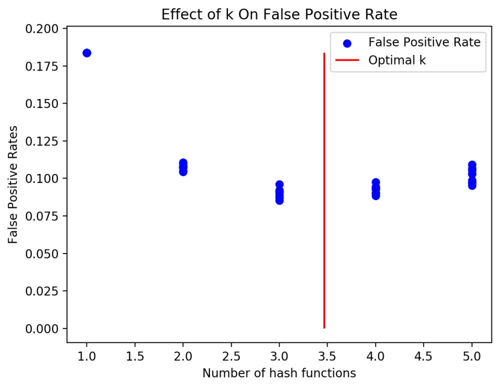
$$(1 - e^{-kn/m})^k = e^{k \ln(1 - e^{-kn/m})}$$

and minimizing $k \ln(1 - e^{-kn/m})$ with respect to $k$. Taking a derivative and setting it equal to 0 tells us that

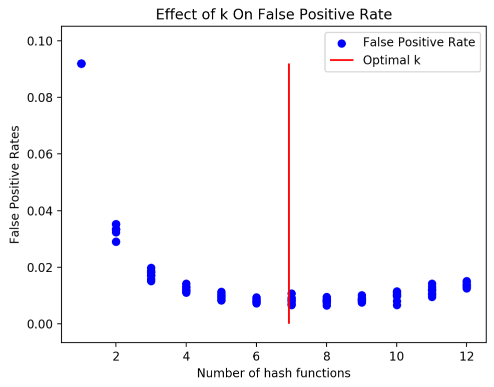
$$\begin{aligned} 0 &= \frac{d}{dk} k \ln(1 - e^{-kn/m}) \\ &= \ln(1 - e^{-kn/m}) + \frac{kn}{m} \frac{e^{-kn/m}}{1 - e^{-kn/m}} \end{aligned}$$

which apparently is equal to 0 when $k = \frac{\ln(2)m}{n}$, and the authors in [1] claim that it's also a global minimum. I have it on good authority that at least one of the authors of that paper is pretty good at what he does, so I'm going to take this fact for granted. Thus, the optimal number hash functions to use is $k = 0.6931\left(\frac{m}{n}\right)$.

We confirm this experimentally by implementing a Bloom filter for varying values of $k$ and plotting the false positive rate:



(a) $m = 5000, n = 1000$



(b) $m = 10000, n = 1000$

Figure 1: False positive rate of a Bloom filter, plotted as a function of the number of hash functions

For larger values of $\frac{m}{n}$, the plot becomes too flat to display meaningfully.

Given a budget of $b = \frac{m}{n}$ bits per element in our key set, we can plug in the optimal value of $k$ and get a false positive rate of

$$(1 - e^{-\ln(2)b(1/b)})^{\ln(2)b} = (1 - e^{-\ln(2)})^{\ln(2)b}$$

which we will write as $\alpha^b$ for simplicity.

## 3 Learned Bloom Filter

We now describe the structure of the Learned Bloom Filter. For most index structures the goal of the hash function is to minimize collisions between keys, but in the case of a Bloom filter, the goal of the hash function is to distinguish between keys and non keys. In an ideal world, we would use a hash function that somehow managed to hash every member of the key set to 0 and every other element in the universe to 1. This would allow us to perfectly determine if a query is in the key set with no false positives or false negatives and with only two bits. Unfortunately, this is clearly not a realistic setup to hope for. We can get as close as possible by training a machine learned oracle to classify the elements of our universe by whether or not they exist in the key set. To preserve the no false negatives guarantee, we would then run all of our keys through our learned function and insert those that are labelled negative into a backup filter. This process can hopefully significantly reduce the number of elements that we insert into our backup filter, which would mean we can reduce the size of our backup filter while keeping a low false positive rate.

## 3.1 Defining the Structure

A Learned Bloom Filter $(f, \tau, B)$ on a set of keys $S$ and a universe $U$ has a learned classifier $f : U \to [0, 1]$ and a backup filter $B$. The classifier has a threshold value $\tau$ where, given an element $u$ in our universe, our learned function classifies $u$ as a member of the key set if $f(u) \geq \tau$. The backup filter $B$ stores all elements $k \in S$ where $f(k) < \tau$, or the classifier reports a false negative. On a query $q$, the Learned Bloom Filter returns $f(q) \geq \tau \vee B(q)$, where $B(q)$ is the backup filter's report on whether $q$ was inserted into it.

To initialize a Learned Bloom Filter, we first need training data for our machine learned oracle. This is usually in the form of the positive key set $S$ and some set of negative keys $\mathcal{U}$, where our data set is

$$\mathcal{D} = \{(x_i, y_i = 1) | x_i \in S\} \cup \{(x_i, y_i = 0) | x_i \in \mathcal{U}\}$$

We then toss our training data into some machine learning model $f$ that for a query $x$ returns a confidence level $f(x) \in [0, 1]$ that can be interpreted as the probability that $x$ is in $S$. As mentioned previously, if this confidence level is above the threshold $\tau$, we say the classifier returns positive, otherwise negative. If we query $f$ with a key $s$ and the learned function returns negative, we insert $s$ into our backup filter. Thus, if our learned function has a false negative rate $F_n$ on the set of keys, we insert a total of $nF_n$ keys into our backup filter.

This model is fairly flexible to different machine learning models, with the only requirement being that the output is a real number between 0 and 1 representing confidence. For example, in [3], they suggest using a neural network trained to minimize the log loss function defined as

$$\mathcal{L} = \sum_{(x,y) \in \mathcal{D}} y \log f(x) + (1 - y) \log(1 - f(x))$$

Other examples of models that output a confidence level are Random Forest and similar Bayesian learning methods. An example of a model that could NOT be used is logistic regression classification, since it simply outputs whether or not a query is on the "positive" side of the function it has learned. However, as we show later, using such a binary classifier isn't necessarily bad, it just doesn't give the flexibility of adjusting $\tau$.

## 3.2 False Positive Rate

The analysis of the false positive rate of a Learned Bloom Filter is much trickier than that of the standard Bloom filter, because in this case we have to take into account the limitations of the learned function. Suppose the universe of elements is the integers $[0, 10000)$, the set of keys are a random set of 20 integers in $[10, 100)$, and the set $\mathcal{U}$ is a set of 20 integers chosen uniformly at random from the rest of the universe. The probability that no element of $\mathcal{U}$ is in the range $[0, 10)$ is 98%, so it seems reasonable for the classifier to conclude that integers less than 100 are in the key set, while integers greater than 100 are not. If this is the case and our query set is precisely the integers $[0, 10)$, we would get a 100% false positive rate. Intuitively, this is because the query set is not representative (or looks "dissimilar" to) the training data.

There may be a silver lining to this apparent cloud in the characteristics of a machine learned oracle. If the query set is very representative of the training data, our oracle will actually perform very well, and in an extreme case given the perfect test set the oracle may be a perfect classifier. However, we can't really hope for this to happen all the time, so we need some way to measure the false positive rate of our classifier that is reliable enough to expect that our measurement is indicative of the false positive rate over the query set.

For a Learned Bloom Filter, we define the empirical false positive rate $F_p$ over a set $\mathcal{T}$ where $\mathcal{T} \cap S = \{\}$ as the number of false positives on elements in $\mathcal{T}$ divided by the size of $\mathcal{T}$.

We also define the false positive rate with respect to a distribution $\mathcal{D}$ of queries over $U \backslash S$ for a Learned Bloom Filter to be

$$\Pr_{y \sim \mathcal{D}}(f(y) \geq \tau) + \Pr_{y \sim \mathcal{D}}(1 - f(y))\mathbb{E}[F(B)]$$

where $F(B)$ is the false positive rate of our backup filter after inserting only false negatives from $f$ and computed as described earlier.

Also as mentioned earlier, the value $F(B)$ is technically a random variable, but since the false positive rate is concentrated around its expectation, and we can easily compute it given the false negative rate of $f$ and the size of our bloom filter, we simply write

$$\Pr_{y \sim \mathcal{D}}(f(y) \geq \tau) + \Pr_{y \sim \mathcal{D}}(1 - f(y))F(B)$$

As long as our test set $\mathcal{T}$ is drawn from the same distribution $\mathcal{D}$ as our query set $\mathcal{Q}$, we can use the empirical false positive rate over $\mathcal{T}$ to estimate what the false positive rate of our Bloom filter will be over the query set. Thus, the empirical false positive rate of our Bloom filter is

$$F_p = \frac{\sum_{t \in \mathcal{T} \sim \mathcal{D}^n} 1 - (1 - f(t))(1 - B(t))}{n}$$

Where $\mathcal{T}$ consists of $n$ elements drawn independently from the distribution $\mathcal{D}$. We can show that with high probability the empirical false positive rate is very close to the true false positive rate:

**Theorem 1.** *Suppose we have a Learned Bloom filter $(f, \tau, B)$, a test set $\mathcal{T}$, and a query set $\mathcal{Q}$ drawn from the same distribution $\mathcal{D}$. Let $X$ be the false positive rate of our Learned Bloom filter on $\mathcal{T}$ and $Y$ be the false positive rate on $\mathcal{Q}$. Then*

$$\Pr[|X - Y| \geq \epsilon] \leq 2e^{-|\mathcal{T}|\epsilon^2/2} + 2e^{-|\mathcal{Q}|\epsilon^2/2}$$

*[2]*

*Proof.* Let $a$ be the probability that a non key $u$ drawn from $\mathcal{D}$ is erroneously classified as positive by $f$, and $b$ be the false positive rate of the backup filter. We can view our number of false positives $X|\mathcal{T}|$ as the sum of the results of $|\mathcal{T}|$ independent trials, each equalling 1 with probability $a+(1-a)b$. By the Chernoff bound,

$$\Pr[X|\mathcal{T}| \geq (a + (1-a)b + \epsilon)|\mathcal{T}|] \leq e^{-2|\mathcal{T}|\epsilon^2}$$

and

$$\Pr[X|\mathcal{T}| \leq (a + (1-a)b + \epsilon)|\mathcal{T}|] \leq e^{-2|\mathcal{T}|\epsilon^2}$$

Dividing through in the inequality by the size of the sample $|\mathcal{T}|$ and combining the two as an absolute value gives

$$\Pr[|X - (a + (1-a)b)| \geq \epsilon] \leq 2e^{-2|\mathcal{T}|\epsilon^2}$$

By a similar line of reasoning,

$$\Pr[|Y - (a + (1-a)b)| \geq \epsilon] \leq 2e^{-2|\mathcal{Q}|\epsilon^2}$$

So combining the two bounds we get

$$\Pr[|X - Y| \geq 2\epsilon] \leq 2e^{-2|\mathcal{T}|\epsilon^2} + 2e^{-2|\mathcal{Q}|\epsilon^2}$$

If we wish to do a change of variables, we can replace all instances of $\epsilon$ with $\frac{\epsilon}{2}$ to get

$$\Pr[|X - Y| \geq \epsilon] \leq 2e^{-|\mathcal{T}|\epsilon^2/2} + 2e^{-|\mathcal{Q}|\epsilon^2/2}$$

□

Thus, the probability that our empirical false positive rate differs from the true false positive rate over our query set by more than $\epsilon$ is exponentially small in the size of the smaller of the two sets. In the context of our earlier example, this result tells us that the probability that our entire query set $\mathcal{Q}$ is in the interval $[0, 10)$ is exponentially small. For the rest of this paper, we write the empirical false positive rate of our learned function as $F_p$ and the false negative rate $F_n$ and treat them as constants given the trained learned function.

Notably, the Learned Bloom filter falters when this result cannot be used, such as when the query distribution cannot be known a priori and we cannot determine the distribution to draw our test set from. In the field of machine learning, if the training set (in this case, both the set that our model is trained on and the test set $\mathcal{T}$) is not representative of the test data (in this case, $\mathcal{Q}$), we call this situation "unfortunate" and simply acknowledge that our model may not generalize well to novel data or may perform poorly on an adversarial test set.

## 3.3 Size of $f$

We may be curious to think about what kinds of trade offs we can expect to see if we allocate more or less limited memory to the learned function $f$ instead of the backup filter. In particular, we should confirm that using a Learned Bloom filter is actually better than just a regular Bloom filter.

Suppose for a key set of size $n$ we have a budget of $bn$ bits for our backup filter. Furthermore, let the size of the learned function $|f| = z$. For a key set of size $n$ the backup filter only needs to hold $nF_n$ keys, so the number of bits stored per key is $\frac{b}{F_n}$. The false positive rate of our Learned Bloom Filter is thus $F_p + (1 - F_p)\alpha^{b/F_n}$.

If we use all of our memory, the combined bits used for our learned function and our backup filter, and simply turn it all into a Bloom filter of size $bn + z$, this Bloom filter uses $\frac{bn+z}{n} = b + \frac{z}{n}$ bits per key and has a false positive rate of $\alpha^{b+z/n}$. Thus, for our Learned Bloom Filter to perform better than a standard Bloom filter, we wish to find $z$ such that

$$F_p + (1 - F_p)\alpha^{b/F_n} \leq \alpha^{b+z/n}$$

Since $\alpha < 10$ so $\log(\alpha) < 0$, this simplifies to

$$\log_\alpha \left(F_p + (1 - F_p)\alpha^{b/F_n}\right) \geq b + \frac{z}{n}$$

This gives us an upper bound on the complexity of the learned function relative to the number of keys. Such a bound is important because it tells us that at a certain point using more and more fancy learned functions with millions of parameters on a training set with 1000 elements is not actually worth it.

## 4 Sandwiched Learned Bloom Filter

In the Learned Bloom filter the false positives from $f$ are reported unchecked by any structure. To reduce the false positive rate, we may wish to increase $\tau$ to only report positives when the model is "pretty sure" that the query is positive. However, making this change also affects the false negative rate. In particular, if $\tau$ is too high, many elements in the key set will probably be classified as not in the key set, and we would have to insert more elements into our backup filter, thus increasing the false positive rate of our backup filter. One possibility proposed by Mitzenmacher in [2] is to use an initial Bloom filter even before the learned function to first filter out some negatives from our query set.

All positive hits to this initial filter are then passed on to the learned function, and negatives from the learned function are queried in the backup filter. The initial filter stores all elements in the key set, while the backup filter in this case still only holds false negatives given by the learned function. The goal is to remove more false positives in the beginning so we can make the backup fairly small with a higher false positive rate. This structure of a learned function between two slices of Bloom filter was hungrily named a "Sandwiched Learned Bloom Filter."

## 4.1 Defining the Structure

A Sandwiched Bloom Filter $(f, \tau, b_1, b_2)$ on a set of keys $S$ where $n = |S|$ and a universe $U$ has an initial filter $A$ of size $b_1 n$, a learned function $f : U \to [0, 1]$, and a backup filter $B$ of size $b_2 n$. As before, we assume that the initial and backup filters choose the optimal number of hash functions given their insertion capacity. On a query $q$, the Sandwiched Bloom Filter returns $A(q) \wedge (f(q) \geq \tau \vee B(q))$.

As with the Learned Bloom filter, we can use any machine learning classifier that outputs a probability and train it on a data set consisting of our keys and some nonkey elements of the universe.

## 4.2 False Positive Rate

If we have a total budget of $b$ bits per key for our bloom filters, to be split into $b_1$ and $b_2$, we wish to compute the optimal values of $b_1$ and $b_2$. As before, given a Bloom filter with a budget of $b$ bits per element in our key set, we write the false positive probability as $\alpha^b$.

The initial filter stores $n$ elements total, so its false positive rate is $\alpha^{b_1}$. The false positive rate of the learned function is $F_p$, computed empirically as before. Since we have a total of $nF_n$ false negatives from our learned function and the size of our backup filter is $b_2 n$, it stores $\frac{b_2}{F_n}$ bits per key inserted. Thus, the false positive rate of an entire Sandwiched Bloom Filter is

$$\alpha^{b_1}(F_p + (1 - F_p)\alpha^{b_2/F_n})$$

which is given by the nature of the output $A(q) \wedge (f(q) \geq \tau \vee B(q))$. If we wish to minimize the false positive probability we can rearrange our expression to get [2]

$$F_p \alpha^{b_1} + (1 - F_p)\alpha^{b/F_n}\alpha^{b_1(1-1/F_n)}$$

and take the first derivative with respect to $b_1$ to get

$$F_p\alpha^{b_1} + (1 - F_p)\left(1 - \frac{1}{F_n}\right)\alpha^{b/F_n}\alpha^{b_1(1-1/F_n)} = 0$$

when

$$\alpha^{b_2/F_n} = \frac{F_p}{(1 - F_p)\left(\frac{1}{F_n} - 1\right)}$$

or

$$b_2 = F_n \log_\alpha \frac{F_p}{(1 - F_p)\left(\frac{1}{F_n} - 1\right)}$$

Which means that given the quality of the learned function ($F_p$ and $F_n$) the bits per key in the backup filter is constant no matter how many keys we have. This seems pretty reasonable, since the backup filter is just there to support the learned function. As long as the learned function doesn't change, we don't need to improve the quality of the backup. Instead, we can simply reduce the number of elements the learned function needs to deal with by increasing the size of the initial filter.

## 4.3 Size of $f$

Again, we may wish to consider what size of an oracle is worth it. If $|f| = z$ and we have a budget of $bn$ bits for both initial and backup filters, we have an improvement on a single Bloom filter if

$$a^{b_1}(F_p + (1 - F_p)a^{b_2/F_n}) \leq a^{b+z/n}$$

which we can solve for a bound on $z$. We may instead wish to compare the false positive probability of our Sandwiched Bloom Filter with that of one without the learned function (i.e. the buns of a burger without the patty).

Suppose we have a tandem Bloom filter with an initial filter and a backup filter. We insert all $n$ keys into both, because there is no longer any notion of false negatives as with the learned function. For a query to be declared positive, it must pass through both initial and backup filters. If we have $x$ bits for the initial filter and $y$ bits for the backup filter for $x + y = bn + z$, we get a false positive if both filters erroneously declare that the query is a match. Thus, the false positive rate of our tandem Bloom filter is $\alpha^{x/n+y/n} = \alpha^{b+z/n}$.

Admittedly I feel rather foolish in this moment but it makes sense that the false positive rate would be the same as with a single bloom filter, since we can simply concatenate the two arrays stored in our two layers and we now have a regular Bloom filter with the combined size. This argument can be extended to splitting a regular Bloom filter into more than just two independent subfilters.

## 5 Implementation Design and Pitfalls

We used a data set taken from Kaggle of Airbnbs in New York City (`https://www.kaggle.com/dgomonov/new-york-city-airbnb-open-data`). Our classification task was determining whether a listing is located in Manhattan, as we inserted all listings with location Manhattan into our structures. The features we trained on are latitude, longitude, room type, price, minimum nights, number of reviews, and days available per year. A priori we thought including the latitude and longitude as features would make the learning task too easy, since most non-Manhattan listings are in Brooklyn and it seems like the classifier can just draw a line down the East River to classify them correctly. However, training on just the longitude and latitude yielded higher false positive rates than on all features. An outline of our implementation is in the appendix.

For our learned function, we used a neural net with one input layer, three intermediate layers, and one output layer. We randomly split the key set into used and unused keys, and randomly split the rest of the data into the non key training data, the empirical false positive testing data, and the query set. We trained with 100 epochs and batches of size 20. Given the trained model, we would change the value of $\tau$ to see its effects on the false positive rate. For all data structures, we ran 10 trials of each set of parameters, and marked the theoretical false positive rate on the plots with a cross.

For the Learned Bloom Filter, we plotted the false positive rate against the size of the backup filter. The implementation is pretty straightforward and the results aren't so surprising

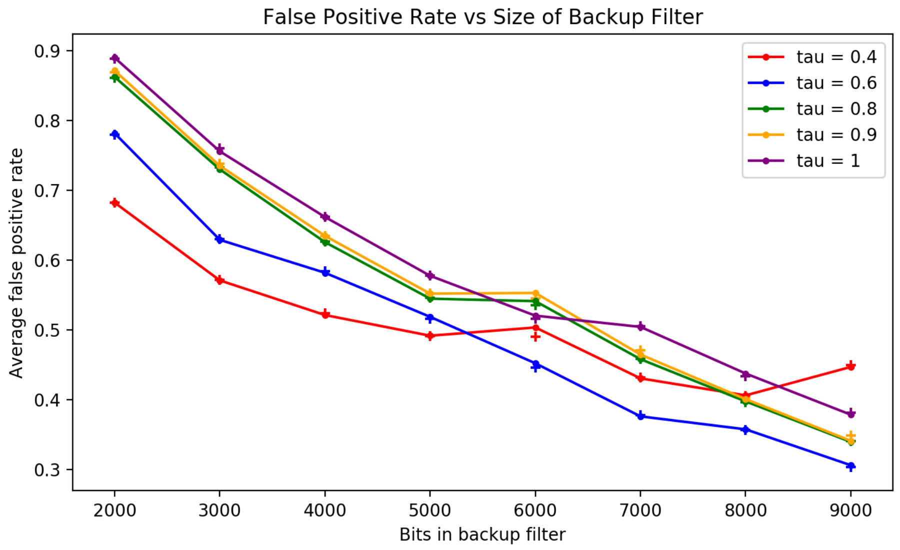


Figure 2: Learned Bloom Filter false positive rate, $n \approx 4000$

We can see that with the trivial learned function $\tau = 1$, our Learned Bloom Filter is effectively a regular Bloom filter and doesn't perform very well compared to to a nontrivial learned function. Either $\tau = 0.4$ or $\tau = 0.6$ performs best, depending on the size of the backup filter. The former is somewhat surprising until you realize that when $\tau = 0.4$ performs best the false positive rate is even higher than a half, so the results don't mean much.

Implementing the Sandwiched Bloom Filter was quite different than the Learned Bloom Filter. Depending on the false positive and false negative rates, it's possible that optimal $b_1$ given by the equation is negative. Unfortunately this is an issue that came up a lot, so to avoid dividing by 0 and other such errors, we set a default lower bound on the number of hash functions to 0 and the number of bits in the Bloom filters to 1 and restricted $b_2$ to be at most $b$. This also means we can't simply set $\tau = 1$ to see the performance with a trivial classifier, because then $F_n = 1$ and the expression for $b_2$ involves dividing by 0.

We plot the false positive rate of our implementation of the Sandwiched Bloom Filter against the budgeted total number of bits per key.

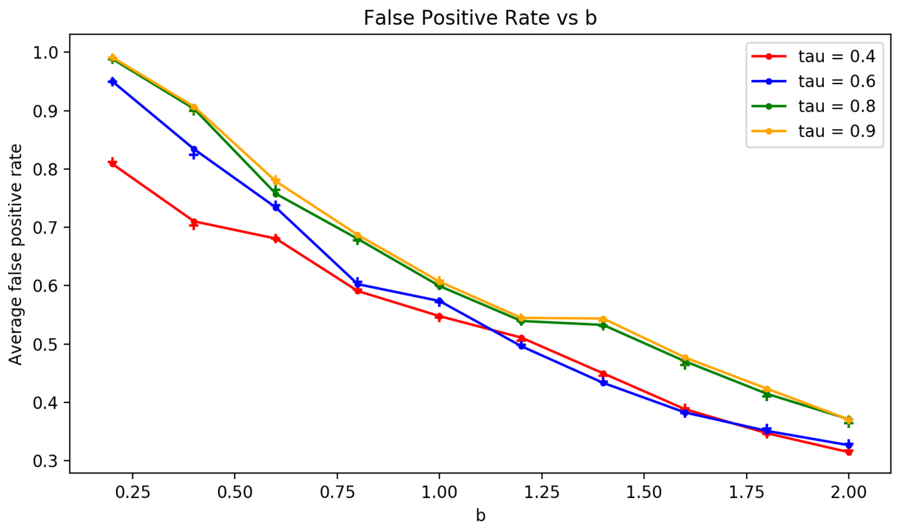


Figure 3: Sandwiched Bloom Filter false positive rate, $n \approx 4000$

We can compare the performance of the Learned Bloom Filter to the Sandwiched Bloom Filter by comparing the plots at $b = x$ to $m = 4000x$ and see that the Sandwiched Bloom filter performs slightly better. In both cases $\tau = 0.6$ appears to be optimal.

## 6 Future Work

For future work, we could solve for the best values of $F_p$ and $F_n$ and look at methods to tune the learned function and $\tau$ to achieve those optimal values. This would require a lot more experience with machine learning that we have and is thus out of the scope of this paper.

We could also apply the analysis involving the size of the learned function to see when one of the Learned Bloom Filter or Sandwiched Bloom Filter is better than the other.

## References

[1] Broder, A., Mitzenmacher, M. and Mitzenmacher, A.B.I.M., 2002. Network applications of bloom filters: A survey. In Internet Mathematics.

[2] Mitzenmacher, M., 2018. A model for learned bloom filters and optimizing by sandwiching. In Advances in Neural Information Processing Systems (pp. 464-473).

[3] Kraska, T., Beutel, A., Chi, E.H., Dean, J. and Polyzotis, N., 2018, May. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data* (pp. 489-504). ACM.

# 7 Appendix

Here are simplified signatures of the relevant classes used to implement the data structures reviewed in this paper:

`class Learned_Bloom_Filter`

- `learned_function`
- `backup_filter`
- `def set_learned_function(f)`
- `def train_learned_function(train_feats, train_labels, layers)`
- `def insert_many(key_IDs, key_feats)`
- `def query_many(query_feats, query_keys)`

`class Sandwiched_Bloom_Filter`

- `initial_filter`
- `learned_function`
- `backup_filter`
- `def set_learned_function(f)`
- `def train_learned_function(train_feats, train_labels, layers)`
- `def init_filters(b, n)`
- `def insert_many(key_IDs, key_feats)`
- `def query_many(query_feats, query_keys)`

`class nn_oracle`

- `oracle`
- `tau`
- `def train(train_dat, train_labels, layers)`
- `def predict(x_test)`

`class NN`

- `model`
- `def fit(layers)`
- `def predict(x_test)`

`class bloom_filter`

- `hasher`
- `bits`
- `seeds`
- `def insert_many(key_arr)`
- `def query_many(key_arr)`

Our implementation of the structures is logically identical to the way they are described in this paper.