Project Report

Name: Sathish Vijay

Course: Algorithmic Machine Learning and Data Science

NetId: svk319

Title of Paper : A model for learned bloom filters and optimizing by sandwiching

Summary/Learning:

One of the main learning from this paper is the relatively new programming paradigm called 'Learning With Predictions' where machine learning is used with traditional algorithms.

The author initially explains about a specific type of bloom filters called learned bloom filters in the paper "The Case for Learned Index Structures" where ML based function 'f' is trained on the input data for which the membership query has to be answered using the bloom filter. Instead of traditional hash functions-based Bloom filter, a ML based function 'f' is used for decision making here. He then states the objective of the current paper which aims to formally model such type of Learned Bloom filters, provide certain guarantees and guidelines for modelling it and he optimizes it by introducing a "Sandwiched model of Learned Bloom filter".

- Firstly he defines what is false positive rate of traditional bloom filter .
- Then he derives the Expectation for false positive rate 'ρ'.
- He then uses Chernoff bound to upper bound the false positive rate.
- He then does the same for the same for learned bloom filter.
- He then makes a series of formal definitions and theorems related to the learned bloom filters.
- In section 4, he discusses the size of the learned function 'f' in terms of bits required per key ζ/m, where 'm' is total elements in the universal set of the bloom filter for which we have to answer the membership query and ζ is the total bits required to represent the function 'f'.

What are Bloom filters?

- A bloom filter is **a probabilistic data structure that is based on hashing**. It is extremely space efficient and is typically used to add elements to a set and test if an element is in a set. Though, the elements themselves are not added to a set. Instead, a hash of the elements is added to the set.

- Internally, Bloom filters use a bit array of size m and k hash functions, which each map a key to one of the m array positions (see Figure9(a)). To add an element to the set, a key is fed to the k hash-functions and the bits of the returned positions are set to 1. To test if

a key is a member of the set, the key is again fed into the k hash functions to receive k array positions. If any of the bits at those k positions is 0, the key is not a member of a set. In other words, a Bloom filter does guarantee that there exists no false negatives, but has potential false positives.

The primary standard theoretical guarantee associated with a Bloom filter is the following. Let y be an element of the universe such that $y \notin S$, where y is chosen independently of the hash functions used to create the filter. Let ρ be the fraction of bits set to 1 after the elements are hashed. Then

$$\Pr(y \text{ yields a false positive}) = \rho^k.$$

For a bit in the Bloom filter to be 0, it has to not be the outcome of the kn hash values for the n items. It follows that

$$\mathbf{E}[\rho] = 1 - \left(1 - \frac{1}{m}\right)^{kn} \approx 1 - \mathrm{e}^{-kn/m},$$

and that via standard techniques using concentration bounds (see, e.g., [11])

$$\Pr(|\rho - \mathbf{E}[\rho]| \geq \gamma) \leq \mathrm{e}^{-\Theta(\gamma^2 m)}$$

Bloom Filter Architectures

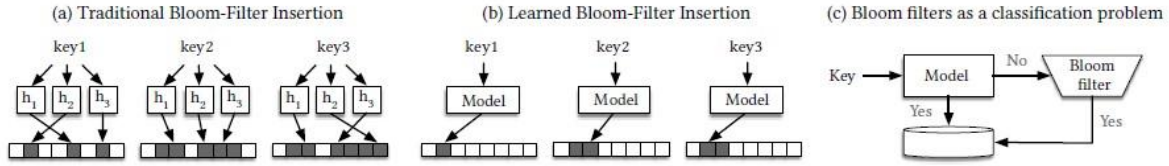

Figure 9: Bloom filters Architectures

What is a Learned Bloom Filter?

Like a standard Bloom filter, Learned Bloom filter provides a compressed representation of a set of keys K that allows membership queries. (We may sometimes also refer to the keys as elements.) Given a key y, a learned Bloom filter always returns yes if y is in K, so there will be no false negatives, and generally returns no if y is not in K, but may provide false positives. What makes a learned Bloom filter interesting is that it uses a function that can be obtained by "learning" the set K to help determine the appropriate answer; the function acts as a pre-filter that provides a probabilistic estimate that a query key y is in K.

Definition 1 *A* learned Bloom filter *on a set of positive keys $\mathcal{K}$ and negative keys $\mathcal{U}$ is a function $f : U \to [0, 1]$ and threshold τ, where U is the universe of possible query keys, and an associated standard Bloom filter B, referred to as a backup filter. The backup filter holds the set of keys $\{z : z \in \mathcal{K}, f(z) < \tau\}$. For a query y, the learned Bloom filter returns that $y \in \mathcal{K}$ if $f(y) \geq \tau$, or if $f(y) < \tau$ and the backup filter returns that $y \in \mathcal{K}$. The learned Bloom filter returns $y \notin \mathcal{K}$ otherwise.*

Definition 2 *A* false positive rate on a query distribution $\mathcal{D}$ *over $\mathcal{U} - \mathcal{K}$ for a learned Bloom filter (f, τ, B) is given by*

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

where $F(B)$ is the false positive rate of the backup filter B.

While technically $F(B)$ is itself a random variable, the false positive rate is well concentrated around its expectations, which depends only on the size of the filter $|B|$ and the number of false negatives from $\mathcal{K}$ that must be stored in the filter, which depends on f. Hence where the meaning is clear we may consider the false positive rate for a learned Bloom filter with function f and threshold τ to be

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

where the expectation $\mathbf{E}[F(B)]$ is meant to over instantiations of the Bloom filter with given size $|B|$.

Size of Learned Function

We assume a total budget of *bm* bits for the backup filter, and $|f| = \zeta$ bits for the learned function. If $|K| = m$, the backup Bloom filter only needs to hold $m*F_n$ keys, and hence we take the number of bits per stored key to be b/F_n. To model the false positive rate of a Bloom filter that uses j bits per stored key, we assume the false positive rate falls as α^j.

The false positive rate of a learned Bloom filter is $F_p + (1 - F_p)\alpha^{b/F_n}$. This is because, for $y \notin \mathcal{K}$, y causes a false positive from the learned function f with probability F_p, or with remaining probability $(1 - F_p)$ it yields a false positive on the backup Bloom filter with probability α^{b/F_n}.

A comparable Bloom filter using the same number of total bits, namely $bm + \zeta$ bits, would have a false positive probability of $\alpha^{b+\zeta/m}$. Thus we find an improvement using a learned Bloom filter whenever

$$F_p + (1 - F_p)\alpha^{b/F_n} \leq \alpha^{b+\zeta/m},$$

which simplifies to

$$\zeta/m \leq \log_\alpha \left(F_p + (1 - F_p)\alpha^{b/F_n}\right) - b,$$

where we have expressed the requirement in terms of a bound on ζ/m, the number of bits per key the function f is allowed.

Learned model vs Sandwiched model

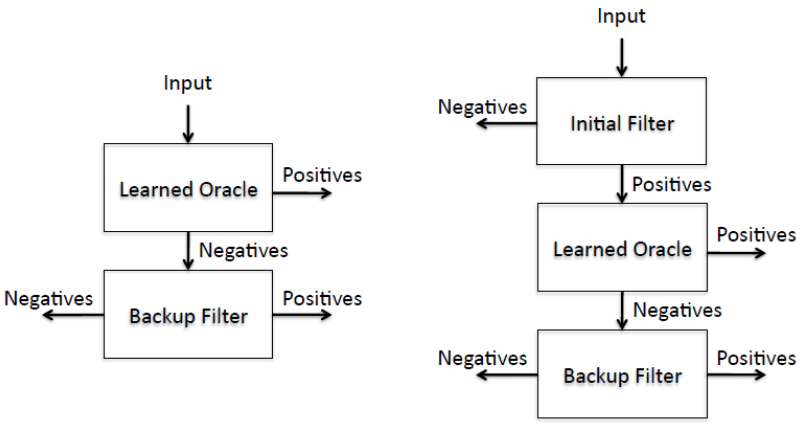

Figure 1: The left side shows the original learned Bloom filter. The right side shows the sandwiched learned Bloom filter.

As α, F_p, F_n and b are all constants for the purpose of this analysis, we may optimize for b_1 in the equivalent expression

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

The derivative with respect to b_1 is

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

This equals 0 when

$$\frac{F_p}{(1 - F_p)\left(\frac{1}{F_n} - 1\right)} = \alpha^{(b-b_1)/F_n} = \alpha^{b_2/F_n}. \quad (1)$$

This further yields that the false positive rate is minimized when $b_2 = b_2^*$, where

$$b_2^* = F_n \log_\alpha \frac{F_p}{(1 - F_p)\left(\frac{1}{F_n} - 1\right)}. \quad (2)$$

Research Idea:

While the paper elaborates on the above stated areas, we aim to optimize the learned bloom filters in two ways.

- Reduce the false positive rate 'ρ' which increases the accuracy of the bloom filters.
- Reduce the number of bits 'ζ' required to represent the function 'f'.

General observation:

The range of the input data(say in input range[0,10000] we want to predict 'y' which can be '0' or '1') can affect the number of bits required to store the weights of trained parameters in logistic regression or simply put the learned function 'f'. The range of the input data is related to the scale of the features, and the scale of the features can impact the scale of the weight values.

In logistic regression, the model assumes a linear relationship between the features and the target variable. The weights assigned to each feature determine the strength of the relationship. If the input features have a large range, it might result in larger weights in the model. Larger weights require more bits to be accurately represented.

The impact of the input data range on the number of bits required for storage is particularly relevant when considering numerical precision and the potential for overflow or underflow. If the range of the input data is very large, it might be necessary to use a higher precision data type (e.g., 64-bit floating-point numbers) to prevent loss of precision during computations. On the other hand, if the range is limited, lower precision data types may be sufficient.

In practice, it's common to standardize or normalize the input features to have zero mean and unit variance. This process can mitigate issues related to the scale of the features, making it easier to choose an appropriate representation for the weights.

Solution for research ideas proposed:

While the paper discusses the learned function 'f' and memory in terms of bits used to represent it, it does not specify an optimized means or model to adopt for learning or training the function 'f' which has impacts on the above two research ideas. As per the paper, the function 'f' is modeled using RNN.

Here we propose to use spline logistic regression to learn the function 'f'. So basically what we mean by this is that, the membership set of the bloom filter for instance say, as per the last paragraph in page 5 of the paper

"set consists of 500 random elements from the range [1000, 2000] and 500 other random elements from the range [0, 1000000). Our learned Bloom filter has f(y) >= τ for all y in [1000; 2000] and f(y) < = τ otherwise."

So let the range be split up into suitable ranges depending on the knots of spline regression and thereby resulting in subsets of the membership set. We can then learn a set of 'n' SoftMax functions(as we use Spline Logistic regression) which will represent each subset of the membership set which is now divided into smaller subsets and can be represented in the form of piecewise continuous unit step functions. This has greater accuracy, and our claim is that it requires lesser number of bits on the whole while compared to the total number of bits 'ζ' required by the single learned function 'f' as mentioned in the paper.

To Prove

Claim 1:

In this setting lets put forth the statement formally which we aim to prove empirically.

$$\sum_{k=1}^{n} \zeta_k \leq \zeta$$

Where ζ denotes the total bits required to represent the function f of the learned bloom filter as mentioned in the paper and ζ_k represents the number of bits required for the 'k'th model out of 'n' models based on our spline logistic regression. For 'n' number of subsets of the original membership set i.e. for 'n' learned functions. In short we have to find (n-1) knots or partition points for the input data so as to optimize the total number of bits, provided the false positive rate is less than or equal to the original model as mentioned in the paper. For this ,In the paper, 'The case for Learned Index Structures' , the dataset used was from Google's transparency report which seems to be a proprietary dataset of Google. Hence we need some more time to gather the benchmark dataset to implement our proposed model based on Spline logistic regression and then we ought to compare the results of the paper.

Claim 2:

Even if we are unable to prove the above equation for all types of membership sets, if it easy to discern the fact that by having separate learned functions for each subset of the total membership set, we can overfit to the data in a better way and thereby reduce the false positive rate of the learned bloom filter for each subset. In other words

$Max_j \rho_j \leq \rho$

where ρ denotes the false positive rate of the model in the paper and ρ_j denotes the false positive rate of the j^{th} model(among n models) as per our design. **The reason why it is lesser than the false positive rate ρ of the learned bloom filter because the total sample set for each of the n models is always lesser than or equal to the total sample set of the learned bloom filter leading to lesser false positives, assuming that equal number of bits are set to 1 after the elements are hashed**.

Challenges and conclusion:

Due to time shortage, it was difficult to get the same benchmark dataset used in the papers to implement our model and compare the results for claim 1 empirically.

Alternative approaches/learnings:

Logistic regression is commonly used for binary classification problems, but when dealing with a range of integer data, especially if there are non-linear relationships, decision tree-based models can be more suitable.

1. **Random Forest:**
    - **How it works:** Random Forest is an ensemble learning method that builds multiple decision trees and merges their predictions. Each tree is trained on a subset of the data and makes independent predictions.
    - **Handling integers:** Random Forest can handle integer inputs effectively, capturing non-linear relationships and interactions between features.
2. **Gradient Boosted Trees (GBT):**
    - **How it works:** GBT builds decision trees sequentially, with each tree trying to correct the errors of the previous ones. It combines weak learners (shallow trees) to create a strong model.
    - **Handling integers:** GBT is also well-suited for integer data. It excels in capturing complex relationships and is less prone to overfitting.

Ensemble Methods:

- Both Random Forest and GBT are ensemble methods, combining multiple models to improve overall performance.
- They are robust to noise and tend to generalize well, making them suitable for a variety of datasets.

Considerations:

- **Feature Importance:** These models can provide insights into feature importance, helping you understand which features contribute the most to the predictions.
- **Hyperparameter Tuning:** It's crucial to tune hyperparameters for optimal performance. Common parameters include tree depth, number of trees, and learning rate for GBT.

Decision Trees vs. Logistic Regression:

- Decision trees naturally handle non-linear relationships, while logistic regression assumes a linear relationship between features and the log-odds of the target variable.
- Logistic regression may struggle to capture complex patterns present in integer data.

In summary, when dealing with a range of integer data and aiming for accurate predictions with non-linear relationships, Random Forest and Gradient Boosted Trees are strong contenders. Experimentation and tuning are essential to find the best model for your specific dataset.

To model a range of integers [−R,R] with binary labels using logistic spline regression, you can use a technique called logistic regression with splines or generalized additive models (GAMs). GAMs allow for flexibility in modeling non-linear relationships between features and the target variable. Here's a simplified explanation:

1. **Basis Splines:**
    - Basis splines are a type of spline functions used to model non-linear relationships. They involve breaking the range of a variable into intervals and fitting a polynomial within each interval.
2. **Generalized Additive Model (GAM):**
    - A GAM is an extension of logistic regression that allows for non-linear relationships by combining multiple smooth functions (splines) of the predictor variables.

<u>Sparse range of input data:</u>

If the samples we are interested in are sparse within the given range [−R,R], and we want to model the relationship effectively, we may consider a more adaptive approach. One way to handle sparse data within a range is by using a localized modeling technique, such as kernel density estimation or localized regression. Here's an outline of how we might approach this:

1. **Kernel Density Estimation (KDE):**
    - KDE estimates the probability density function of a random variable. In your case, it can be used to estimate the distribution of the integer data within the given range.
2. **Localized Regression:**
    - Use a regression model that adapts to the local density of the data. Locally weighted scatterplot smoothing (LOWESS) or kernel regression are examples of methods that give more weight to nearby points when estimating the regression function.

```
3. import numpy as np
4. import matplotlib.pyplot as plt
5. from sklearn.neighbors import KernelDensity
6. from statsmodels.nonparametric.smoothers_lowess import lowess
```

```
import statsmodels.api as sm

# Generate example data
np.random.seed(123)
R = 10
n = 100
x = np.random.uniform(low=-R, high=R, size=n)
y = np.random.binomial(1, p=1 / (1 + np.exp(-0.5 * x + 0.2 * x**2)))

# Kernel Density Estimation (KDE)
kde = KernelDensity(bandwidth=1.0, kernel='gaussian')
kde.fit(x[:, None])

# Generate test points within the range
x_test = np.linspace(-R, R, 1000)
log_density = kde.score_samples(x_test[:, None])

# Plot KDE
plt.figure(figsize=(10, 5))
plt.plot(x_test, np.exp(log_density), label='KDE')

# Localized Regression (LOWESS)
lowess_smoothed = lowess(y, x, frac=0.3, it=3)
plt.scatter(x, y, alpha=0.5, label='Data')
plt.plot(lowess_smoothed[:, 0], lowess_smoothed[:, 1], color='red',
label='LOWESS')

# Logistic regression for comparison
x_design = sm.add_constant(x)
logit_model = sm.GLM(y, x_design, family=sm.families.Binomial())
result = logit_model.fit()
y_pred = result.predict(x_design)
plt.plot(x, y_pred, linestyle='dashed', label='Logistic Regression')

plt.legend()
plt.show()
```

3. **Interpretation:**
    - The KDE helps us understand the overall density of your data, and localized regression techniques capture non-linear relationships in regions with more data.

Experiment with the bandwidth and fraction parameters to fine-tune the models according to the **Kernel Density Estimation (KDE):**

- KDE estimates the probability density function, and the range of predicted output is not constrained to a specific range. The output represents the density of the data at different points within the range. It is not directly interpretable as a probability or binary outcome.

2. **Localized Regression (LOWESS):**
   - LOWESS provides smoothed predictions for the binary target variable. The output will be continuous values between 0 and 1, representing estimated probabilities. However, LOWESS itself does not enforce a strict range.

3. **Logistic Regression:**
   - Logistic regression models the probability of a binary outcome. The output will be in the range [0,1] as it represents the estimated probability of the positive class.

It's important to note that the three models (KDE, LOWESS, and Logistic Regression) serve different purposes in the example:

- KDE provides a smooth estimate of the density of the data.
- LOWESS gives a smooth fit of the binary target variable, providing estimated probabilities.
- Logistic Regression models the binary outcome with predicted probabilities in the [0,1] range.