Data Management Algorithm for Decision Making
02360003

Home Assignment 1 - Dry Part

Ido Tausi - 214008997
Afek Nahum - 214392706
Or Mutay - 206918633

Problem 1: Bloom Filter

1.

Description of the designed Bloom Filter:

- Number of elements to be inserted (n): 100
- Number of hash functions (k): 3
- Desired false positive rate: 0.1.

To achieve the desired false positive rate according to those values, we can use the formula from the lecture, which is $Prob[false\ positive] = \left(1 - e^{\frac{-k \cdot n}{m}}\right)^k$ when m is the number of bits, k is the number of hash functions, and n is the number of elements to be inserted. Therefore we'll require:

$$\left(1 - e^{\frac{-k \cdot n}{m}}\right)^k \leqslant 0.1$$

$$\left(1 - e^{\frac{-3 \cdot 100}{m}}\right)^3 \leqslant 0.1$$

$$1 - e^{\frac{-300}{m}} \leqslant \sqrt[3]{0.1}$$

$$e^{\frac{-300}{m}} \geqslant 1 - \sqrt[3]{0.1}$$

$$\frac{-300}{m} \geqslant ln\left(1 - \sqrt[3]{0.1}\right)$$

$$m \geqslant \frac{-300}{ln\left(1 - \sqrt[3]{0.1}\right)} = 480.832$$

From here we can conclude that the minimum number of bits needed is 481, in order to achieve the desired false positive rate of 0.1.

2.

Now we would express the optimal number of hash functions (k) for a Bloom filter as a function
of the size of the Bloom filter (m) and the number of elements inserted (n).

We will derive and find the minimum of the false positive function. According to the lecture,

the function is: $f(k) = \left(1 - \left(1 - \frac{1}{m}\right)^{-k \cdot n}\right)^k$

Now, we'll assume that m and $k \cdot n$ are big enough, and we will make this estimation:
$\left(1 - \frac{1}{m}\right)^{-k \cdot n} \approx e^{\frac{-k \cdot n}{m}}$

Therefore the function can be estimated with this approximation
$f(k) = \left(1 - \left(1 - \frac{1}{m}\right)^{-k \cdot n}\right)^k \approx \left(1 - e^{\frac{-k \cdot n}{m}}\right)^k$

To ease the calculation process, we'll look at the $log(f(k))$ function. since $\ln$ is a monotone function, the minimum point of $log(f(k))$ will share the same k value as $f(k)$.
$log(f(k)) = log\left(\left(1 - e^{\frac{-k \cdot n}{m}}\right)^k\right) = k \cdot log\left(1 - e^{\frac{-k \cdot n}{m}}\right)$

$$\frac{\partial log(f(k))}{\partial k} = log\left(1 - e^{\frac{-k \cdot n}{m}}\right) + \frac{1}{1 - e^{\frac{-k \cdot n}{m}}} \cdot \left(\frac{n}{m}\right) e^{\frac{-k \cdot n}{m}} \cdot k =$$

$$log\left(1 - e^{\frac{-k \cdot n}{m}}\right) + \frac{e^{\frac{-k \cdot n}{m}}}{1 - e^{\frac{-k \cdot n}{m}}} \cdot \frac{nk}{m}$$

We will demand $\frac{\partial log(f(k))}{\partial k} = 0$, substitute $t = e^{\frac{-k \cdot n}{m}}$, and solve for t:

$$log(1 - t) + \frac{t}{1 - t} \cdot\!- log(t) = 0$$

$$(1 - t)log(1 - t) - t\ log(t) = 0$$

We can see that for $t = \frac{1}{2}$ the expression $(1 - t)log(1 - t) - t\ log(t)$ is equal to 0 since $1 - t = 0.5 = t$ in this case.
Therefore the optimal k value is received when:

$$t = \frac{1}{2} \Leftrightarrow e^{\frac{-k \cdot n}{m}} = \frac{1}{2} \Leftrightarrow \frac{-k \cdot n}{m} = log\left(\frac{1}{2}\right) \Leftrightarrow k = \frac{m}{n} log(2)$$

3.

When the Bloom Filter size m is not sufficiently large, the assumption we made in the previous question: $\left(1 - \frac{1}{m}\right)^{-k \cdot n} \approx e^{\frac{-k \cdot n}{m}}$, no longer holds because m needs to be large.

Considering this, we **will not recommend** increasing the value of k, since it represents the number of hash functions used in the bloom filter, and so, using many hash functions on a relatively small bit-array will cause (on average) many bits to be equal '1', and thus the amount of false positive results would most likely go up in this scenario.

We recommend decreasing k to reduce redundant hashing and minimize false positives under the constrain (m).

4.

a. No, we'll give an example in which B doesn't represent $A_1 \cup A_2$:

Let $m = 3,\ k = 1$, and our hash function would be $h(n) = n \bmod 3$.

Also, take $A_1 = \{1\}, A_2 = \{2\}$ and so $A_1 \cup A_2 = \{1, 2\}$.

By definition of the hash function, we would get $B_1 = 010,\ B_2 = 001$ (the left bit is the 0-index and the right bit is the 2-index).

We can now perform bitwise AND on the bloom filters: $B = B_1 \& B_2 = 000$

But the bloom filter that matches $A_1 \cup A_2$ is not 000, but 011, and thus we showed a contradiction to the claim.

b. No, we'll give an example in which B doesn't represent $A_1 \setminus A_2$ in the requested way:

Let $m = 3,\ k = 1$, and our hash function would be $h(n) = n \bmod 3$.

Also, take $A_1 = \{1\}, A_2 = \{2\}$ and so $A_1 \setminus A_2 = \{1\}$.

By definition of the hash function, we would get $B_1 = 010,\ B_2 = 001$ (the left bit is the 0-index and the right bit is the 2-index).

We can now perform bitwise AND on the bloom filters: $B = B_1 \& B_2 = 000$

Now, if we check the element 1 which is in $A_1 \setminus A_2$ with B, we would **not** get a positive answer for a membership query on B, although 1 is in $A_1 \setminus A_2$. We got a false negative, therefore we got a contradiction.

Problem 2: Counting unique items

1.

In the python notebook.

2.

The results we got after 50 experiments are:

```
Maximum Error: 1.2054006097110133
Minimum Error: 0.03892684647534139
Average Error: 0.40473586077514084
```

According to the results, we can say that the bloom filter-based estimator performs well, with a low average error of 0.405 and a maximum error of 1.205, indicating that it provides close estimates of the real unique values on average.
However, this estimator is still unsuitable if we need precise counts, as it tends to underestimates due to false positives (inherent in bloom filters). Its accuracy can be further improved by optimizing the bloom filter's parameters.

3.

a.

In the python notebook.

b.

```
=====Adding the item: 5=====
Hash value #0 for item 5: 1
Hash value #1 for item 5: 12
Hash value #2 for item 5: 4
Hash value #3 for item 5: 11
=====Adding the item: 12=====
Hash value #0 for item 12: 6
Hash value #1 for item 12: 15
Hash value #2 for item 12: 5
Hash value #3 for item 12: 8
=====Adding the item: 7=====
Hash value #0 for item 7: 7
Hash value #1 for item 7: 6
Hash value #2 for item 7: 2
```

```
Hash value #3 for item 7: 1
=====Adding the item: 5=====
Hash value #0 for item 5: 1
Hash value #1 for item 5: 12
Hash value #2 for item 5: 4
Hash value #3 for item 5: 11
=====Adding the item: 10=====
Hash value #0 for item 10: 0
Hash value #1 for item 10: 5
Hash value #2 for item 10: 7
Hash value #3 for item 10: 2
=====Adding the item: 7=====
Hash value #0 for item 7: 7
Hash value #1 for item 7: 6
Hash value #2 for item 7: 2
Hash value #3 for item 7: 1
=====Adding the item: 12=====
Hash value #0 for item 12: 6
Hash value #1 for item 12: 15
Hash value #2 for item 12: 5
Hash value #3 for item 12: 8
=====Adding the item: 7=====
Hash value #0 for item 7: 7
Hash value #1 for item 7: 6
Hash value #2 for item 7: 2
Hash value #3 for item 7: 1
=====Adding the item: 10=====
Hash value #0 for item 10: 0
Hash value #1 for item 10: 5
Hash value #2 for item 10: 7
Hash value #3 for item 10: 2
=====Adding the item: 12=====
Hash value #0 for item 12: 6
Hash value #1 for item 12: 15
Hash value #2 for item 12: 5
Hash value #3 for item 12: 8
```

c.

```
True count of unique items: 4
FM Sketch estimate: 4.5
```

d.

	Bloom Filter	**FM Sketch**
Memory usage	16 bits	$k \cdot log(log(n))$ bits. n - the largest number in the domain. k - the number of hash functions.
Accuracy of estimate	Average Error on 50 experiments: 0.4047358	FM Sketch error: 0.5 Distance of 12.5% from the true count.
Computational efficiency	**Initialization:** $O(k \cdot n)$ n - number of elements inserted to the Bloom Filter. k - the number of hash functions. For each element of n elements, we apply k hash functions and setting the bit to 1. Therefore, $O(k \cdot n)$ computational efficiency for initialization of Bloom Filter. **Estimation:** $O(m)$ m - size of bit-array. For each entry of the bit-array, we check and count if the bit is 0. Therefore, $O(m)$ computational efficiency for estimating unique items using Bloom Filter.	**Initialization:** $O(k \cdot nlog(n))$ n - number of elements inserted to the Bloom Filter. k - the number of hash functions. For each element of n elements, we apply k hash functions and counting the number of trailing zeros of each value. Therefore, $O(k \cdot nlog(n))$ computational efficiency for initialization of FM Sketch. **Estimation:** $O(k)$ k - the number of hash functions. For each counter of the max trailing zeros of hash function, we calculate the average of all 2^{value}. Assuming the computation of each element is $O(1)$, the computational efficiency is $O(k)$ for estimating unique items using FM Sketch.

e.

For estimating the number of distinct elements in large datasets, we recommend to use the approach of **FM Sketch**. This is because when dealing with large datasets, the bloom filter would have to use a big bit array in order to estimate the accurately the number of distinct elements in the database, but, according to our analysis and the lecture, FM sketch is less demanding in memory usage but yields similar results in terms of accuracy, while having similar computational efficiency as well. Thus, when considering all these different aspects, we would recommend the use of FM Sketch over bloom filters when working with large data sets.

Problem 3: MinHash

1.
In the python notebook.

2.
The statement is false, we'll show a contradicting example:
$U = \{0, 1, 2\},\ A = \{1\},\ B = \{1, 2\} \implies A \subset B$
Let P_1 be a permutation of U: $P_1 = \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix}$
Now, the MinHash signature would be of length=1, because we have one hash function, which is the permutation P_1.
According to the MinHash signature definition, A's MinHash signature will be: $[1]$.
Meanwhile, B's MinHash signature will be: $[2]$.
Therefore, their MinHash signatures don't match perfectly even though $A \subset B$, and thus the statement in the question is **false**.

3.
The statement is **true.**
We learned in the lecture that MinHash similarity is calculated as such: $\frac{count}{k}$, where k is the number of hash functions, and $count$ is the counter of the similar elements in the sets according to the counting in the MinHash algorithm.
Also, the Jaccard Similarity is defined as such: $\frac{|A \cap B|}{|A \cup B|}$, and the MinHash similarity is supposed to estimate it using $\frac{count}{k}$.

Now, we can consider the law of large numbers in probability:
The average of the results (Min Hash similarity) obtained from a large number of independent random samples (k hash functions) converges to the true value (Jaccard similarity).
Each hash function gives a sample, and using more samples reduces randomness. This makes the MinHash estimate more accurate and closer to the actual Jaccard similarity.

4.

a.

```
MinHash Signature for A: [5, 2, 5]
MinHash Signature for B: [9, 2, 4]
```

b.

The Jaccard similarity is: $\frac{|A \cap B|}{|A \cup B|} = \frac{|\{2, 8, 9\}|}{|\{2, 4, 5, 8, 9, 11, 13\}|} = \frac{3}{7} = 0.4285$

```
Jaccard Similarity for A and B: 0.42857142857142855
```

c.

```
Jaccard Similarity approximation unsing Minhash, k=3: 0.3333333333333333
Jaccard Similarity approximation unsing Minhash, k=5: 0.4
```

The MinHash similarity approximate the Jaccard similarity and approaches the true Jaccard similarity as k increases.
With $k = 3$, the approximation deviates more from the true Jaccard similarity compared to $k = 5$.
The differences might occur as fewer hash functions can result in higher variance in the estimate and a larger k reduces variance and improves accuracy.

Problem 4: Locality Sensitive Hashing (LSH)

1.

According to the lecture, the probability of two sets that will get the same hash is dependent on the Jaccard Similarity s, the size of the MinHash signature which is t. This probability also depends on the number of bands, b. Also, the size of each band is r, which means $\frac{t}{b} = r$.

$$Prob[\textit{two sets will get the same hash}] = 1 - \left(1 - s^r\right)^{t/r} \overbrace{=}^{t/r=b} 1 - \left(1 - s^r\right)^b$$

The values of t, b and r directly effect the probability of collisions. Those parameters effect the accuracy of the LSH algorithm by changing the chance for collisions in the same hash bucket, but they may also impact the accuracy of the colliding pairs.

We will describe how varying b and r affects the probability of finding pairs with high Jaccard similarity:
b: increasing b will increase the probability for collisions of 2 sets (according to the $Prob[\textit{two sets will get the same hash}]$). However, it may reduce precision as more pairs pass through, including ones with low similarity (since there is no guarantee that 2 sets with the same hash have high Jaccard Similarity).

r: inversely to parameter b effect, increasing this parameter will increase the precision by emphasizing the weight of high Jaccard similarity, thus passing pairs with high similarity. However, it will reduce the probability for 2 sets to get the same hash according to the $Prob[\textit{two sets will get the same hash}]$.

2.

We want pairs with Jaccard similarity of at least 0.8 to be detected, so we want to get a high probability of collisions for those kind of pairs.

In order to do that, we will try to get: $1 - \left(1 - 0.8^r\right)^b \to 1$.

Setting r to be relatively small and b to be large, for example $r = 2,\ b = 10$ $(t = r{*}b = 20)$ will result a very close result to 1 for $Prob[\textit{two sets with } s > 0.8 \textit{ will get the same hash}]$. This will ensure that nearly all pairs with $s > 0.8$ are detected.