HW1 - Data Management Algorithms for Decisions Making

Shaza Abdallah - 323032474 Shir Mey-Tal - 208638791

December 14, 2025

1 Problem 1: Bloom Filter

1.1

We know from the lecture that the false positive probability of a Bloom filter is

$$\Pr[\mathrm{FP}] = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k}.$$

we require:

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} \leq 0.1.$$

after algebraic simplification we get:

$$m \geq \frac{1}{1 - \left(1 - 0.1^{1/k}\right)^{\frac{1}{kn}}}.$$

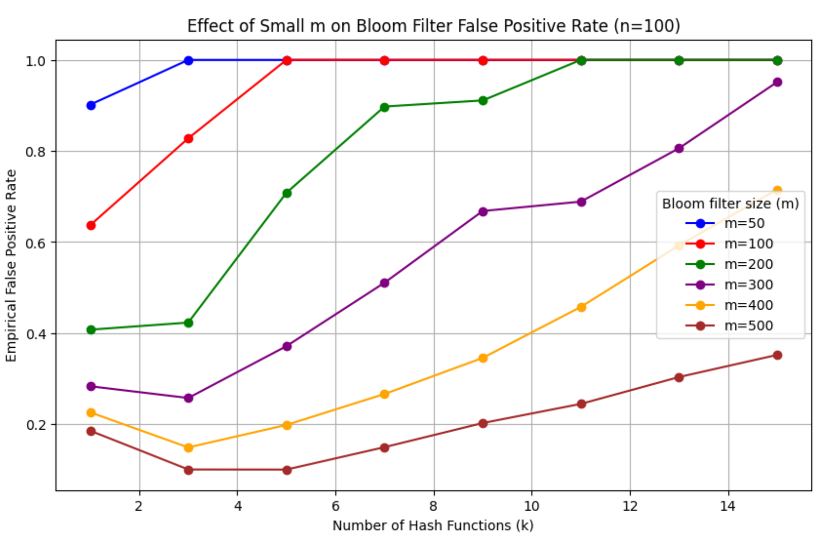
Substituting $n = 1000$ and $k = 5$ gives approximately

$$m \approx 5016.3$$

m is the number of bits so it is an integer, therefore the minimum nimber of bits is:

$$m = 5017 \text{ bits}.$$

1.2



From the plot, we observe that when the memory m is small, increasing the number of hash functions k is not recommended as the false-positive rate increases. Using large k causes the Bloom filter to become saturated quickly, which significantly increases the false positive rate. This is visible in the plots: the false positive rate rises for high k .

However, when the memory is larger (for example $m \geq 200$), using too few hash functions is also sub-optimal. In particular, for these larger values of m, choosing $k < 3$ increases the false positive rate. In the simulations, $k = 3$ consistently achieves the lowest false positive rate for $m \geq 200$.

This behavior matches the formula

$$\Pr[\text{FP}] = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k}.$$

As k increases, two opposite effects occur:

- The inner term $(1 - 1/m)^{kn}$ decreases, since we apply more hash functions and thus the entire expression that is raised to the power k increases.
- The entire expression is raised to the power k, decreases as k increases

Because these two effects pull in opposite directions, the false positive probability is not monotonic in k. increasing k is harmful when the memory is limited. Therefore, when m is small, we should decrease k, not increase it but also avoid making k too small, since very small values of k also increase the false positive rate.

1.3

1.3.1 (a)

No, B is not necessarily the same Bloom filter as the one created by inserting all elements of $A_1 \cup A_2$.

Consider the example:

$$A_1 = \{1\}, \qquad A_2 = \{2\},$$

using a 5-bit Bloom filter and the hash function

$$h(x) = x \bmod 5.$$

Then the Bloom filters are:

$$B_1 = [0, 1, 0, 0, 0],$$
$$B_2 = [0, 0, 1, 0, 0].$$

Their bitwise AND is:

$$B = B_1 \wedge B_2 = [0, 0, 0, 0, 0].$$

But the Bloom filter created by inserting all elements of

$$A_1 \cup A_2 = \{1, 2\}$$

would be:

$$B_{A_1 \cup A_2} = [0, 1, 1, 0, 0].$$

Clearly,

$$B \neq B_{A_1 \cup A_2}.$$

1.3.2 (b)

No, B does not accurately represent the set $A_1 \setminus A_2$.

In the same example:

$$A_1 \setminus A_2 = \{1\},$$

but the bitwise AND Bloom filter was

$$B = [0, 0, 0, 0, 0],$$

so querying for element 1 would incorrectly return “not present.”

Thus, the Bloom filter obtained by ANDing B_1 and B_2 does not correctly represent the set difference $A_1 \setminus A_2$.

2 Problem 2: Counting Unique Items

2.1

2.1.1 (a)

If we insert n elements into a Bloom filter of length m using k hash functions, then the probability that a specific bit i remains 0 is:

$$\Pr[\text{bit } i = 0] = \Pr\left[h_k(r) \neq i \ \ \forall r \in L,\ \forall k\right] = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}.$$

Therefore, the expected number of zero bits in the Bloom filter is:

$$c = m \cdot e^{-kn/m}.$$

If we observe c zeros, then solving for n gives:

$$n = \frac{m}{k} \log\left(\frac{m}{c}\right).$$

The average error is 5.7

2.1.2 (b)

(i) Similarly to the above analysis, the probability that bit i is 1 is:

$$\Pr[\text{bit } i = 1] = 1 - \Pr[\text{bit } i = 0] \approx 1 - e^{-kn/m}.$$

Thus, the expected number of one-bits in the Bloom filter is:

$$c = m\left(1 - e^{-kn/m}\right).$$

Solving for n, we obtain:

$$n = -\frac{m}{k} \log\left(1 - \frac{c}{m}\right).$$

(ii) The average error is 6.35

(iii) Theoretically, the two estimators are equivalent, since they are derived from the same expression for the Bloom filter's bit statistics. In practice, our simulations show that their average errors are very similar (about 5–6), which is consistent with the theoretical equivalence. The small difference between them likely comes from randomization and from floating-point precision effects when evaluating logarithms on values very close to 0 or 1. So maybe those minor numerical variations produce slightly different estimates in practice.

2.2

2.2.1 (a)

We did a kind of cheating in this implementation, since the number of bits of the hash function should satisfy $l = O(\log n)$, where n is the number of unique items. Thus, we should have an a priori idea of how many unique items could appear in the list. However, since this is a synthetic toy example and the items are random, we believe that actually counting the number of unique items in order to determine the appropriate value of l is a valid approach. We also considered other approaches. For example, one could take $l = \log m$, where m is the number of items (in our case $m = 100$). However, this choice does not scale well to large datasets, and therefore does not reflect the complexity guarantees we want. Another option is to examine the `generate_random` function: the number of unique items it produces is uniformly distributed between 30 and 100. Thus, the expected number of unique items is approximately 65, and choosing $l = \log_2(70)$ could also be a reasonable heuristic but this is a hard coded approach which we less prefer.

The average error is 9.02

2.2.2 (b)

The analysis is essentially identical: the probability that the r least significant bits are 1 is $1/2^r$, and in expectation we need to observe about 2^r distinct elements before seeing r consecutive ones. Thus, the theoretical behavior is the same as in the trailing-zeros estimator.

The average error is 9.03

2.2.3 (c)

Both estimators are theoretically equivalent, and in our experiments they show very similar average error (approximately 9). However, compared to the previous estimator, they perform worse. This is likely due to their higher variance: the estimator outputs only values of the form 2^r, which makes the estimate less precise.

3 Problem 3: Jaccard Similarity

3.1

The statement is false. Consider the following counter example:

$$U = \{1, 2\}$$

$$A = \{1\}$$

$$B = \{1, 2\}$$

$$k = 1 \text{ (MinHash signature length)}$$

Clearly, $A \subseteq B$. We show a permutation $P1$ where their MinHash signatures differ.

U	A	B	P1
1	1	1	2
2	0	1	1

As the MinHash signatures are the first elements in each group from each permutation, the MinHash signature for A is [1] and the MinHash signature for B is [2], thus different.

3.2

The statement is true. As we saw in the lecture, if we define event A as "encountering a matching permutation element before encountering an unmatching permutation element", then

$$Prob(A) = Jaccard\ Similarity.$$

Therefore if we look at every permutation as an i.i.d experiment, we can estimate the probability of event A as

$$\frac{1}{k}\sum_{i=1}^{k} \mathbf{I}(\text{event A happened in permutation } i).$$

As we have more experiments we will get closer to the true probability of event A, which is the Jaccard similarity. Therefore, as k increases, we make more experiments, and thus get a better approximation of the Jaccard similarity.

3.3

The statement is false. Consider the following counter example:

$$U = \{1, 2\}$$
$$A = \{1\}$$
$$B = \{1, 2\}$$
$$k = 1$$

Clearly, $A \neq B$. We show a permutation $P1$ where their MinHash signatures are the same.

U	A	B	P1
1	1	1	1
2	0	1	2

As the MinHash signatures are the first elements in each group from each permutation, the MinHash signature for A is [1] and the MinHash signature for B is also [1], thus the signatures are the same though the groups are not identical.

3.4

3.4.1 (a)

Denote by On(A) and On(B) the groups of indices of the bits that are set to 1 in BF(A) and BF(B), respectively. The proposed estimator $E(A, B)$ would then be:

$$E(A, B) = \frac{|On(A) \cap On(B)|}{|On(A) \cup On(B)|}.$$

Intuitively, this estimator approximates the Jaccard similarity because the 1's in the BF of A and B are a representation of the objects that are in these groups. Because of that, this estimator holds similar properties to the Jaccard similarity:

When $A = B$, the elements in A turn on exactly the same bits as the elements in B in their respective Bloom filters, hence $On(A) \cap On(B) = On(A) \cup On(B)$, and the value of E is 1, while the Jaccard similarity is also 1.

In a similar fashion, if there are many shared elements between A and B, all of these shared elements turn on the same bits in BF(A) and BF(B) and our estimator would be high.

When similarity between A and B is low, provided m and k were chosen appropriately, we should get a low similarity on our estimator. This would only be an approximation however, as bitwise collisions are relatively common in our setup, and in our estimator we can't factor in the unique combinations of bits each element turns on when using Bloom filters as intended, only the bitwise intersection of the filters.

3.4.2 (b)

The results we obtained:

```
Trial 1: True Jaccard = 0.538, Estimated = 0.792, Overlap = 35
Trial 2: True Jaccard = 0.333, Estimated = 0.659, Overlap = 25
Trial 3: True Jaccard = 0.042, Estimated = 0.514, Overlap = 4
Trial 4: True Jaccard = 0.053, Estimated = 0.605, Overlap = 5
Trial 5: True Jaccard = 1.000, Estimated = 1.000, Overlap = 50
Trial 6: True Jaccard = 0.087, Estimated = 0.610, Overlap = 8
Trial 7: True Jaccard = 0.493, Estimated = 0.787, Overlap = 33
Trial 8: True Jaccard = 1.000, Estimated = 1.000, Overlap = 50
Trial 9: True Jaccard = 0.235, Estimated = 0.673, Overlap = 19
Trial 10: True Jaccard = 0.852, Estimated = 0.939, Overlap = 46

Average absolute error over 10 trials: 0.2946
```

We can see that in higher similarities the estimation is relatively not too far from the true Jaccard, but in lower similarities the estimator overestimates the similarity by a lot. That fits what we expected the results to be, as same Bloom filter bits are activated also for different elements.

3.4.3 (c)

Using MinHash, we are directly approximating the Jaccard similarity, as we have the guarantee that for each permutation, the probability of finding a matching element before finding an element only in one group is the true Jaccard similarity. Therefore in theory by choosing $k = |A \cup B|$ and taking different permutations, we can calculate the true Jaccard similarity.

However, using Bloom Filters we are not approximating the Jaccard similarity directly, instead relying on turned on bits from both filters, and while we have the guarantee that if an element is in both A and B, its bits would be turned on in both, we can still have a lot of bits that are turned on by none matching elements in A and B, thus creating an over approximation of the true Jaccard similarity.

To improve the accuracy of the estimator, we could use Bloom Filters with a larger m (more bits in each filter), which would make it less likely for different elements to turn on the same bits, which should help the over approximation in our estimator.

Another idea is to use the concept of multiple experiments from the MinHash idea and apply it to Bloom Filters - instead of saving one Bloom Filter per group, save multiple, compute the estimator over each pair and average these, to create a more robust estimator, as it is again less likely different elements would turn on the same bits in all filters.

LLM Use

We did use an LLM in this homework, mainly for translating text into LaTeX. Here is an example (and even this section was revised by the LLM :)).

our input: "We did a kind of cheating in this implementation since the number of bits of the hash fanctio should satsify $l = O(\log n)$ where n is the number of unique items. So we should have an idea about how many unique items could be in the list but since this a synthitic toy example and the items are random we think that actually counting rhe number of unique itmes in order to know how much l should be is a validate thing to do. in latex"

LLM output: "We did a kind of cheating in this implementation, since the number of bits of the hash function should satisfy $l = O(\log n)$, where n is the number of unique items. Thus, we should have an a priori idea of how many unique items could appear in the list. However, since this is a synthetic toy example and the items are random, we believe that actually counting the number of unique items in order to determine the appropriate value of l is a valid approach. "

We also used an LLM when working with the trailing estimators. We had an issue with the hash function when truncating its output to l bits, and it helped us correct the implementation (unfortunately we lost the exact prompt).