Antiplagiarism poem system - bit string arrays and bloom filters

Computer-aided simulations lab

Michele Veronesi
Politecnico di Torino
michele.veronesi@studenti.polito.it

***Abstract*—In this report, we present the analysis of an anti-plagiarism system based on the text of the famous Italian poem Divina Commedia. The focus is given to four different methodologies: a set storing all the sentences in section II, a set storing all the fingerprints in section III, a bit strings array in section IV and optimally-designed bloom filters in section V. Finally, in section VI, we summarize the obtained results.**

I. Preliminary study

The software used during our experiments take as input the following parameters.

- The fixed length of a sentence S.
- The absolute file-path to the textual file containing the Divina Commedia.

As output, it produces the required graphs and values for each of the following sections. The performed operations of the raw text are the followings.

1) Produce a list in which each element is a line of the raw text.
2) Remove all the punctuation from the lines: in this way, two words connected with an apostrophe will be considered a unique token, while all the other cases will produce two different tokens separated by a blank space.
3) Convert all the words to lower-case.
4) Split each line, obtaining a list in which each element is a single word (removing empty strings).
5) Roll over the list produced at the previous point a sliding window of size S and stride 1. In this way, we extract the required sentences.

As required, we use $S = 6$, obtaining a total number of distinct sentences equal to 96.532. The average size of each sentence is 100.65 Bytes. Doing a simple multiplication, all the distinct sentences together should weight ~ 9.27 MB.

II. Set of sentences

The extracted sentences, with the procedure described in the previous point, are then stored in a Python set, using the string format. It's size is ~ 13.27 MB, showing that there is an overhead of 4 MB in terms of memory, i.e., an overhead around 30%. This is because the array underlying the Python set is expanded by a factor 2^x when its size is around 2/3 of its capacity (with size we intend the number of elements into the array, while with capacity the actual number of memory locations which compose the data structure).

III. Set of fingerprints

In this section we analyze the benefits and drawbacks of storing the fingerprints of each sentence instead of the strings directly. First of all, by simulation we evaluate b_{exp} defined as the minimum value of bits such that no collisions are experienced when storing all the sentences in a fingerprint set. To do so, we compute the fingerprints of all the sentences with a fixed size $\xi \in [20, 40] \subset \mathbb{N}$. In particular, $\forall \xi$ we perform the following procedure.

1) Compute the fingerprint with dimension 2^{ξ} (i.e., it occupies ξ bits) for each distinct sentence.
2) Store the fingerprints in a Python set.
3) If the number of elements in the set is equal to the number of distinct sentences, then $b_{exp} = \xi$, otherwise peek the next value for ξ and repeat.

We obtain that $b_{exp} = 33$ is the minimum number of bits for the size of the fingerprints such that no collisions among different sentences are experienced.

Now, we compute analytically the number of bit b_{teo} necessary to get a probability $p = 0.5$ of fingerprint collision when storing all the sentences with a fingerprint set, using eq. (1). We obtain $b_{teo} = \log_2(n) = 33$ with n the dimension for the fingerprint function. Hence, it is a very good approximation of b_{exp}.

Finally, using a system with a fingerprint set, using as the size of the fingerprint 33 bits, we obtain a false positive probability in the order of 10^{-5}. To compute it, we divide the number of distinct fingerprints in the set (which in this case is equal to the number of distinct sentences) by the dimension of the fingerprint, i.e., $96.532/2^{33} \approx 1.1 \cdot 10^{-5}$. With the same settings, the size of the fingerprints set is ~ 754.28 KB, which is a huge reduction with respect to the set of sentences.

$$n = \left\lceil \frac{m^2}{2\log(\frac{1}{1-p})} \right\rceil \tag{1}$$

IV. Bit strings array

We now take into account bit strings arrays. They are array of bits (for the sake of simplicity, in the code they are represented as NumPy arrays of boolean values) with size equal to the dimension of the fingerprint. In this way, instead of storing the fingerprint value, to store an element we put equal to one the cell of the bit string array in the position of the fingerprint value. In our settings, the probability of false positive in function of the size of the bit string array is reported in fig. 1.

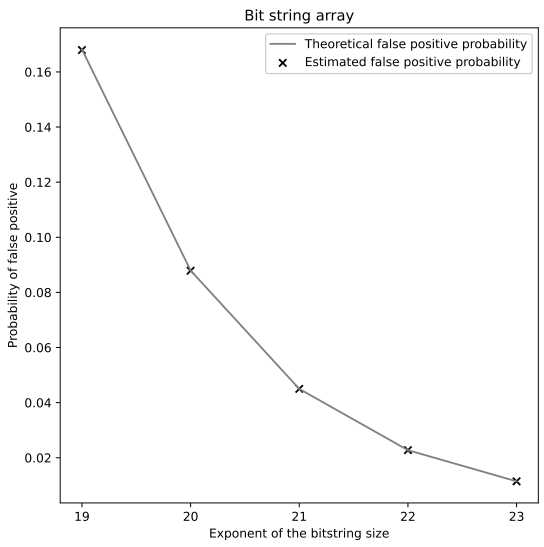

Fig. 1: Probability of false positive w.r.t. bitstring array size.

$$p = 1 - \left(1 - \frac{1}{n}\right)^m \qquad (2)$$

The theoretical false positive probability is computed using the formula in eq. (2), where n is the size of the bit string array, i.e., 2^x, and m is the number of distinct sentences to be stored. Instead, the experimental false positive probability, is computed by using the same technique used for the fingerprints set in section III, using as a dimension for the fingerprint the dimension of the bit string array (i.e., 2^{19}, 2^{20}, . . .).

The main advantage of this technique is obviously the reduced amount of memory used for storing the bit string array instead of a set of fingerprints, while keeping the same false positive probability in function of the fingerprint size. By the way, in our implementation we cannot show the real benefits since the underlying structure is a NumPy array of boolean, therefore using a significantly higher amount of memory.

V. Bloom filters

We consider now bloom filters. They are a simple extension of bit string arrays, where we apply $k > 1$ hash functions instead of one, to each element. Thus, all the values in the array corresponding to the outcomes of those k functions are imposed equal to 1, in order to store a new element. The optimal k in function of the size of the bit string array is reported in fig. 2. Note that this optimization does not take

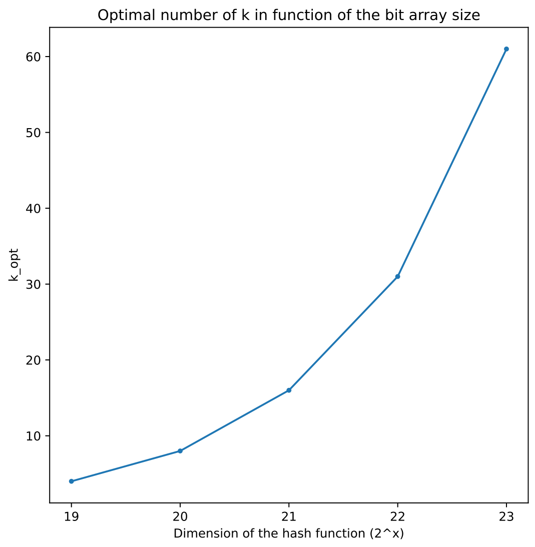

Fig. 2: Optimal number of hash functions in function of memory occupation.

into account the saturation problem, i.e., the speed with which the array becomes full. As you can imagine, this speed is proportional to the parameter k.

Instead, the false positive probability in function of the array size, using the optimal k values, is reported in fig. 3. To obtain

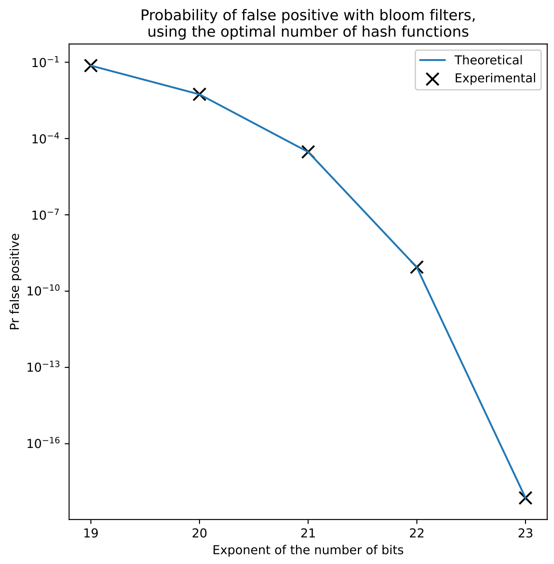

Fig. 3: Probability of false positive using bloom filters with the optimal number of hash functions.

the experimental results, we implement a technique to simulate the behavior of k independent hash functions. Since those functions does not grow on trees like apples, we use the same function over and over but appending an integer shift value to the string passed as input. This machinery works since a good hash function produces an output which is nearly uniform over the output space. To compute the theoretical results, we use the formula in eq. (3), where k is the number of hash functions applied to each sentence. Again, we show that the theory is a good approximation of the expected values.

$$p = \left(1 - \exp\left\{-\frac{k \cdot m}{n}\right\}\right)^k \tag{3}$$

Finally, we perform a study on the theoretical formula for computing the number of elements stored in a bloom filter, reported in eq. (4), where n is the size of the bit array, k is the number of hash functions, N is the number of entries equal to 1 in the array. We report the absolute error of this formula in fig. 4. As you can see, it is proportional with the number of stored elements. This result was expected, since by increasing the number of elements it grows also the number of collisions (i.e., the likelihood that two different elements produce at least one identical hash function output increases).

$$-\frac{n}{k} \cdot \log(1 - \frac{N}{n}) \tag{4}$$

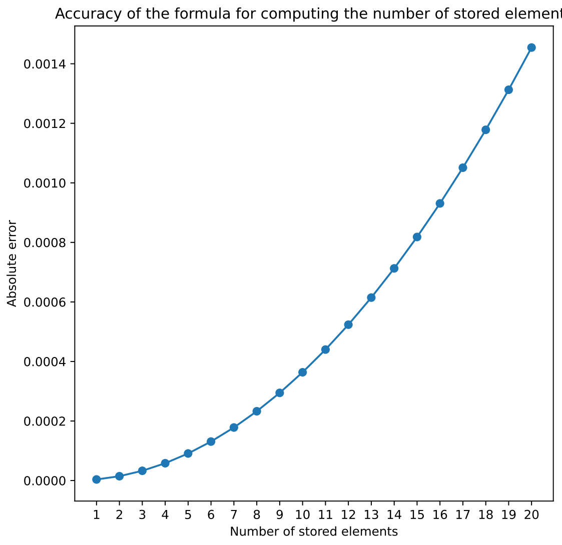

Fig. 4: Absolute error of the formula for computing the number of elements in a bloom filter.

VI. Conclusions

The first solution which stores directly the sentences into the set, it is surely the best in term of false positive, since it is impossible that two different sentences collide. By the way, it is quite memory consuming (~ 13.27 MB for a simple text). Moving forward to the solution which stores the fingerprints into a set, using a size for the fingerprints which occupy 33 bits, thus which do not cause any collision among the sentences of the analyzed text, we can see that the probability of having a false positive is no more zero, but is still acceptable (in the order of 10^{-5}). Instead, the memory occupation drops to ~ 754.28 KB, which is the best result overall. Thirdly, we use bit string arrays. The size of the data structure, in function of the number of bits used, is reported in fig. 5. As you can notice, the implementation which relies on NumPy arrays of boolean is extremely efficient, with a very small overhead. By the way, bit string arrays are not a good solution, since they raise memory occupation by a factor of 10 w.r.t. the set of fingerprints, while worsening the false positive probability (fig. 1). Finally, bloom filters are taken into account. From table I, you can see that this is the best solution overall.

TABLE I: Comparison of the different data structures. For bit string arrays and bloom filters, the number of bits is fixed to 2^{22}.

Data Structure	**Memory**	**Prob. of False Positive**
Set of sentences	13.27 MB	0
Set of fingerprints	754.28 kB	10^{-5}
Bit string array	512 kB	0.02
Bloom filter	512 kB	10^{-10}

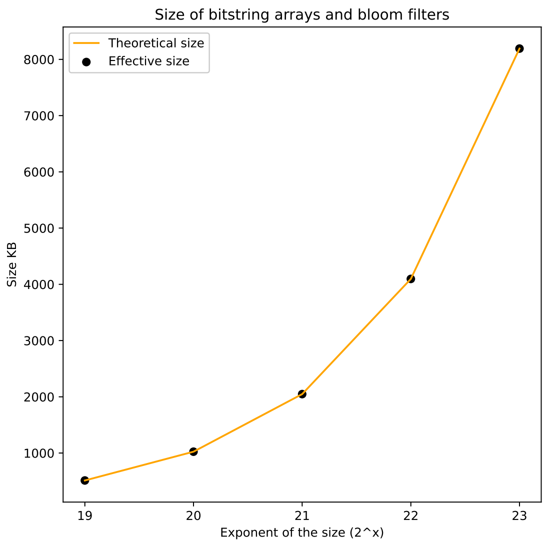

Fig. 5: Comparison of the effective memory occupation against theoretical one, using boolean NumPy array.