Report Assignment 2

Brilli Matteo

INTRODUCTION

All-pairs similarity search is the problem of finding all pairs of records that have a similarity score above the specified threshold. In this project, we have implemented a simhash algorithm to evaluate approximate cosine similarity between two documents from a large collection of files in order to find the near duplicates.

TF-IDF

As a starting point, we have loaded either the News Database or the Enron mail database available on the university cluster, and then filtered the relevant information. This gives us the set of documents to analyze. We can indicate with n the number of documents. Using the functions already implemented within Pyspark it's then very easy to calculate the tf-idf representation of our dataset. A very important property to specify though is the so called `numFeatures` within the `HashingTF` class. Withit the Pyspark implementation of term frequency infact, a raw feature is mapped into an index (term) by applying a hash function. Then term frequencies are calculated based on the mapped indices. This approach avoids the need to compute a global term-to-index map, which can be expensive for a large corpus, but it suffers from potential hash collisions, where different raw features may become the same term after hashing. In order to avoid the chance of collision, while still using the less space possible, we can set `numFeatures` to be equal to the total vocabulary size, i.e. to the number of different words in all documents. Let's call this number d. The value for d can be easily calculated separately, in a different program, for example passing the database to the `CountVectorizer` class implemented in Pyspark which, among other things, also returns the vocabulary size.

SIMHASH

Up to this point we have worked with a $n \times d$ matrix. Our goal is to estimate the cosine similarity between any couple of documents to find all the pairs of documents whose similarity is above a certain threshold. To avoid quadratic complexity, the best solution is to use locality-sensitive hashing (LSH), which will be described better in the next section.

One way to produce a family of LSH functions compatible with the cosine similarity is to compute the SimHash signature of each document. To do this, we consider a family of random hyperplanes, represented by random vectors $r_w = \{-1, +1\}^d$. Two documents are said to be similar if they lie on the same side of an hyperplane, meaning the dot product of the vector representing the document with the hyperplane is positive. In this case return a value of 1, otherwise 0. By repeating this process m times we get a m bit signature for each document, which can be used to estimate the similarity. Obviously, the larger m, the more hyperplanes we consider, the more accurate the estimate.

LSH

We now have a $n \times m$ signature matrix, where each row represents a document, to which we can apply LSH. The intuition is that instead of comparing every pair of elements we just hash them into buckets trying to map to the same buckets all the document whose similarity is above a certain threshold.

To do this, we split each row into p bands of b elements each, so that $m = pb$. Two rows are considered candidate pairs (meaning they have the possibility to have a similarity score above the desired threshold), if the two simhashes have at least one band in which all the bits are identical.

Since the result of the random hyperplane procedure yields a binary signature matrix, we can easily hash the bins using the following trick. For a given row, we want to hash some binary vector like (1,1,0,0,0,1,1,0,1). Instead of creating a b-dimensional tuple to hash, we can instead convert this binary vector to an integer, by simply considering it as the binary number 110001101.

Practically speaking, in the Pyspark implementation we are dealing with an RDD containing the SimHash signatures. Then, through a series of simple map transformations, we can split the signatures and convert each band to an integer. At this point, we need to understand which documents have at least a band in common. To do this, we generate a series of key-value pairs that have as key the index of the band, and whose value contains informations about the id of the document and the integer conversion of the band. We then collect all the informations relative to each band through a `reduceByKey` and finally map all documents having the same integer value to the same bucket.

Probability analysis

Let's now analyze the role of parameters m, p and b. We can see how increasing p gives rows more chances to match, so it lets in candidate pairs with lower similarity scores, while increasing b makes the match criteria stricter, restricting to higher similarity scores.

Now, it's clear that using this approach we will always identify correctly all simhashes with at least $m - p + 1$ equal bits. Depending on p and b though, we will also let in a number of false positives. Let's call p_t the true percent similarity between a pair. The probability that at least one of the band matches is

$$P(\geq 1 \text{ match}) = 1 - (1 - p_t^b)^p$$

We can see in the plot below how tuning p and b changes the probability that we will label two rows as a candidate pair, given they have a true similarity score of p_t.

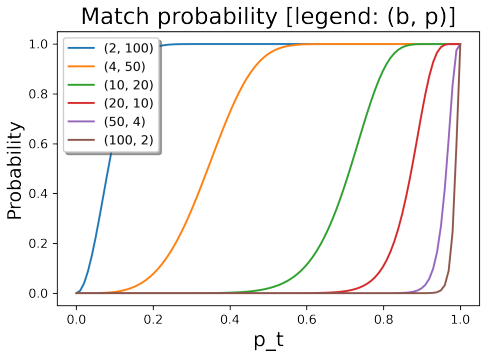

Figure 1. Effect of increasing/decreasing p and b while keeping n fixed

So, given a set of parameters p and b, we can have an idea of how many false positive will be identified by our algorithm. This can have a great impact of the running time of the program, especially if we want to check manually the similarity between the identified documents

If we want to tune our LSH procedure to select more precisely the candidate pairs we should tune p and b while making m as high as we can without increasing the computation time too much.

ANALYSIS

Let us start by reporting some performance figures for different values of the parameters. The following times are measured using the News

dataset on the university cluster. Unless otherwise specified, the programs is executed with 4 workers, one with 16 cores, the others with 2 cores. The first table considers a SimHash size of $m = 128$. The third column measures the number of candidate pairs, while the fourth reports the number of real pairs identified when counting the number of pairs with more than $m - p + 1$ equal bits.

p	seconds	c.p.	r.p
1	8.7	114648	114648
2	8.9	127001	126941
4	9.7	165837	149815
8	11.7	1475743	848783

Table 1. Table for $m = 128$

As we expected, when we increase p we get a bigger fraction of false positives, and this impacts the running time of the algorithm.

Still, it's better to let in too many candidate pairs than too few, because we can always manually check LSH candidate pairs to eliminate false positives, but there's no equivalent trick for false negatives.

Varying hash size

Let us now analyze what happens for varying values of m.

p	s	c.p.	r.p.
1	13.7	114647	114647
2	13.8	123753	123751
4	14.0	124968	124445
8	14.9	162480	157401
16	21.7	1927767	1602001

p	s	c.p	r.p.
1	23.1	114647	114647
2	23.5	119136	118994
4	24.4	121797	121786
8	26.9	137882	136870
16	31.8	162311	161287

Table 2. Tables for $m = 216$ and $m = 512$ respectively

We get a running time proportional to m, and also a decreasing fraction of false positives, demonstrating that the estimate gets indeed more accurate.

Speedup

Let's analyze now how the running time of the algorithm changes when varying the number of workers and cores(keeping $m = 128$ and $p = 4$ fixed):

workers	**cores** (per worker)	**seconds**	**speedup**
1	1	291.4	1
2	1	144.6	2.01
3	1	74.9	3.89
4	1	46.5	6.27
4	2	35.7	8.16
4	4	19.8	15.2

Table 3. Speedup $m = 128$ and $p = 4$

Varying dataset size

To conclude, we can see how the program behaves when changing the dataset size. This analysis was carried on considering the Enron mail dataset. The following data are measured considering again $m = 128$, $p = 4$, and increasing the portion of documents.

Portion of data	**s**	**c.p.**	**r.p.**	**s** (to find r.p.)
1/4	13.2	5662641	4450223	26.5
2/4	21.1	22191280	17404370	104.3
3/4	33.1	49276499	38525512	262.7
4/4	37.9	110260733	84204875	534.2

Table 4. Speedup $m = 128$ and $p = 4$

The main difference with the previous dataset lies in the number of real pairs of duplicate documents. For this very reason, in this case, the measurement of the time taken for the whole algorithm to execute is split in two. In the second column is reported the time taken to identify all the candidate pairs, in the last column instead is reported the time taken to check all pairs of documents to find those with similarity above the threshold. As we can see, this time gets increasily bigger when we have a large number of candidate pairs to check.