

Data Mining: Learning from Large Data Sets - Fall Semester 2015

mmarti@student.ethz.ch
trubeli@student.ethz.ch

October 18, 2015

Approximate near-duplicate search using Locality Sensitive Hashing

In this project we used linear hashing to approximate the similarity between videos, represented by a list of shingles. The first step in our solution was to produce a signature matrix. This matrix is obtained by using a min hash algorithm on each list of shingles. For every i^{th} shingle in a video we pick two random numbers a_i and b_i which are coprime. The hash function is applied by computing:

$$a_i * s_i + b_i \mod n$$

Where s_i is the i^{th} shingle in a video and n is the number of shingles. The procedure for computing the signature matrix is described as follow:

Algorithm 1 Min Hash Algorithm

```
1: procedure MINHASH( $N, K$ ) ▷  $K$  hash function applied on each of the  $N$  shingles
2:   Initialize:
      $w_l \leftarrow \infty, l = 1, \dots, k$ 
3:   for  $i = 1$  to  $n$  do
4:     for  $j = 1$  to  $k$  do
5:       if  $h_j(n_i) < w_j$  then
6:          $w_j \leftarrow h_j(n_i)$ 
```

This signature matrix is then split into b bands of r rows. In order to decrease the number false positive and false negative, we decided to use a combined r -way *AND* and b -way *OR* of hash function on the split matrix. While the original set of hash function was:

$$(d_1, d_2, p_1, p_2)\text{-sensitive}$$

for some p_1, p_2 . The r -way *AND* turns F into a new family F' which consists of vectors of hash functions in F . We therefore obtain a new family of hash functions s.t. each function in F' is:

$$(d_1, d_2, p_1^r, p_2^r)\text{-sensitive}$$

Then, applying a b-way *OR*, we just require that at least one band has to be hashed to the same bucket in order for two videos to be considered similar. Therefore turning our family of hash functions into:

$$(d_1, d_2, 1 - (1 - p_1^r)^b, 1 - (1 - p_2^r)^b)\text{-sensitive}$$

The values of b and r have been tuned accordingly in order to find a sufficient ratio between false positive and false negative. Using values such as we managed to obtain a score of 0.94.

Scripts to simplify local testing

Sequenteially computing results for the test data we were given took quite a lot of time, even on a fast computer. This was due to the fact that the solution ran on only one core instead of all available cores. To simplify and speed up the local testing process we implemented a set of small scripts, written in Ruby, which not only make sure that our mappers and reducers are ran in parallel, but also automatically check the solution and keep track of the solution with the best score.

The scripts support two modes of operation, one for manual runs of the current mapper and reducer and one for continuous testing of different mappers and reducers. The manual approach takes the current mapper and reducer and runs multiple instances of each in parallel to ensure maximum performance. The intermediate sorting is done directly in Ruby. A more efficient approach is the continuous one. Here, we run a separate script which will periodically check a folder for new entries. Entries are added by executing another script, which will create a new folder in the queue and copy the current mapper and reducer into it. The observing script will then find this new folder and run the map-reduce algorithm on it. Once it is done, it checks the outcome of the process using the provided check script and moves the folder from the queue directory to a done directory, where it creates files with all intermediate outputs. Additionally, if the F1 value of the tested instance is higher than the best recorded F1 value, the instance is also copied into a special folder named **best** in the done folder and the maximum F1 value is updated.

This approach allows us to push solutions to be tested faster than they are executed. We can think about parameters that make sense, push them onto the queue and continue working while our results are being computed.