

LSH

Josep de Cid Rodríguez, Adrián Sánchez Albanell

January 2019

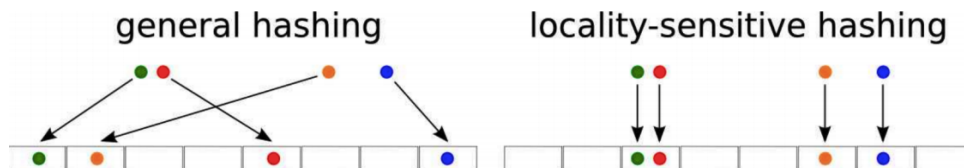
Abstract

In this practice we are going to look for similarity on images representing hand-written digits using a python implementation of a local sensitive hashing technique. We will see its speed and accuracy behaviour when tuning k and m parameters, and compare it with exhaustive search.

1 Task 1: Understanding LSH

The reason to use LSH is to reduce the dimensions on high-dimensional data, which we will use to check similarity between images and find their nearest neighbours.

The main idea is to hash the images into buckets such that similar images are mapped with a high probability to the same bucket aiming to maximize the collision probability for the similar ones, the opposite idea of the general hashing.



In the implementation the LSH is initialized with a list of m dictionaries and $m \cdot k$ randomly selected bits for the hashing. Then we need to hash all images. For each image, we perform m iterations i , hashing the image with a function h_i , which returns a code. We append into the corresponding dictionary bucket the index of the image.

The hashing function h_i flattens the image, choosing the k randomly initialized bits previously and building a code bitstring setting 1 or 0 depending on the comparison between the value given by k and the real value of the corresponding pixel.

To end with the implementation, we have the candidates function, which performs m iterations, checking in each one if the hashed image exists on the dictionaries corresponding to that iteration, returning a set with all the matching images.

We can prove the correct understanding of the given code by doing some experiments with different k and m parameter values and explaining how the execution time and number of candidates vary and why.

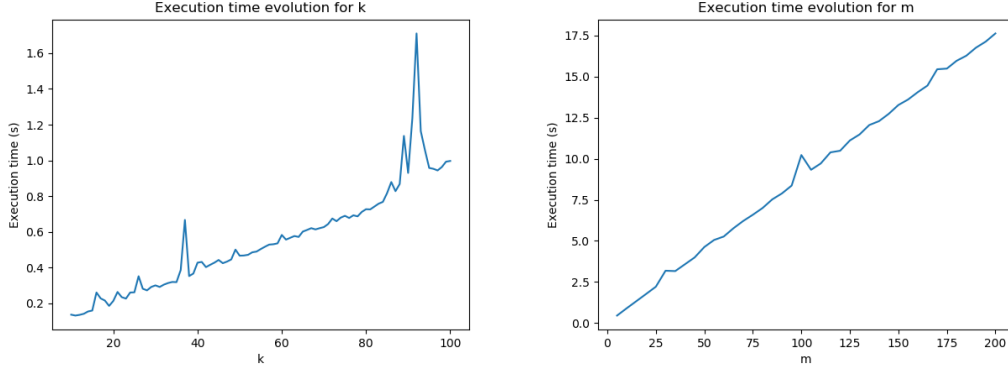


Figure 1: Execution time evolution for parameters k and m .

For both parameters the time increases linear to k and m as we can see in figure 1. This happens due to by increasing these parameters, which changes the number of selected bits and repetitions respectively, increases linearly the number of operations to perform.

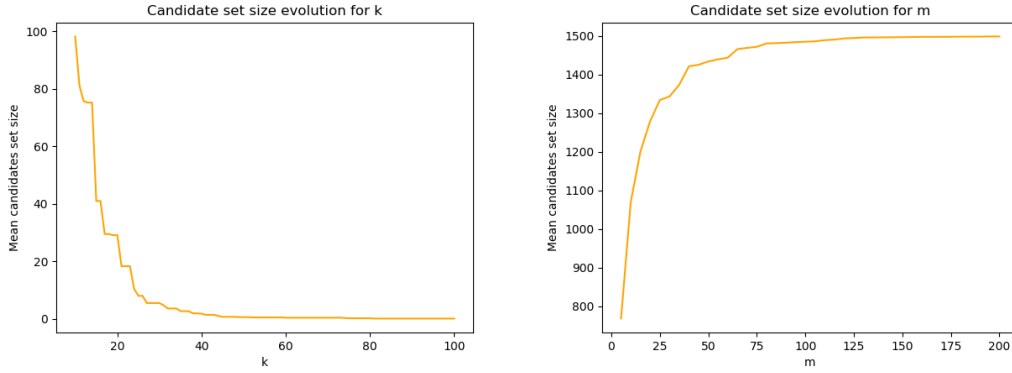


Figure 2: Mean candidates set size evolution for parameters k and m .

The results are the expected as we've seen in the lectures. By stacking a higher number of k hash functions, the probability of collision of similar objects decreases, resulting in much less candidates for a test image.

In the opposite way, when we increase m , the probability of collision increases, given that for each iteration, the result set is built as the union of the previous iterations and the current one, so with high m the set of candidates tends to be equal to the test set.

2 Task 2: Does LSH work?

As the statement says we will start implementing a distance function between two images, which will be the Hamming distance:

$$d(i1, i2) = \sum_{i=1}^d |i1_i - i2_i|$$

Then we will implement the search functions, which will compute the closest image for a given one from the test set. For the brute-force case, it will iterate over all training set images computing the minimum distance to the given image.

By the other side, for the hashing search function, we will implement a very similar function to the brute-force one, only differing on calculating the distances only to the set of candidates instead of doing it with all the test set.

Let's first analyze how the execution time differs in both cases. For small test set sizes the brute-force method takes a higher time to look for the closest result. Why this behaviour? Probably this is given to the time spent to create all the hash tables of the LSH and the small number of times to repeat the brute-force process (10 in the default dataset).

In the figure 3 we can observe how this behaviour is reversed when we increase the test set size, where creating hash tables and using those with LSH method is worth more to repeat a brute-force method n -times, being the time growth much slower.

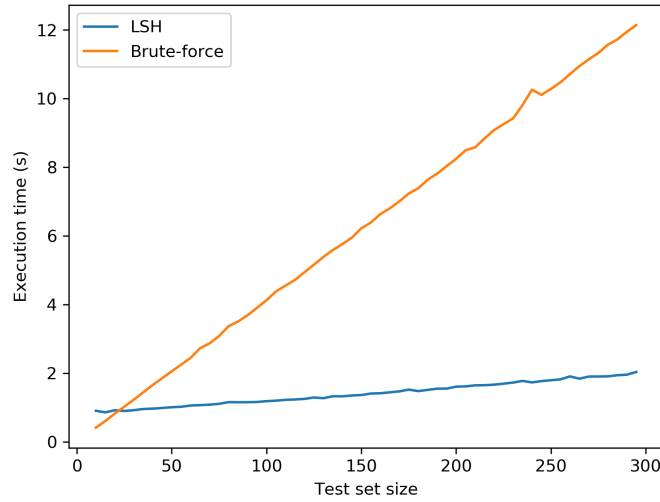


Figure 3: Execution time evolution for testset size

But, does LSH works fine for our case? The brute-force algorithm finds the most similar image as it checks exhaustively the distance to all the possible images. So we will compare the result and distance given by the LSH implementation with different k and m values to the brute-force one.

To do that, we will take 10 images as the test set and execute both algorithms comparing the results as we can see in the table 1 and then extract some conclusions.




















































Image	Brute-force		LSH							
			k=1 ; m=1		k=20 ; m=5		k=50 ; m=5		k=100 ; m=5	
	Nearest	Dist	Nearest	Dist	Nearest	Dist	Nearest	Dist	Nearest	Dist
		52		52		52		52		52
		56		56		56		56	-	-
		60		60		60	-	-	-	-
		64		64		64		64	-	-
		57		57		57		73		57
		30		30		30		30	-	-
		64		110		64		64	-	-
		45		45		45		45	-	-
		52		52		52		65	-	-
		40		40		40		40	-	-

Table 1: Nearest image found and distances with Brute-force algorithm and parametrized LSH

We can extract some conclusions from the results reported in the table. LSH works fine for our problem because there are only a few categories to look for (0..9) and those are pretty different from the others. An important thing to note is that as we increase the parameter k the number of possible candidates decreases and there's a point where no candidates are found and we can't look for the nearest image.

If m is increased, we will have a higher number of candidates (as seen in figure 2), and it will tend to find the nearest image (same as the one found by the brute-force algorithm) that turns into more accuracy, but just because it will check a subset of images that is almost the same as the complete set, spending a higher time to check all images, which is opposite as the strength of the LSH method.

If we set low k and m finds the same number that the image represents but it isn't the same image, it's just another representation of that number. If we set intermediate parameters like $k = 20$ and $m = 5$ we can observe that the images are the same as the obtained with the brute-force method.

Hence, using LSH with properly parameter selection, is so useful to look for the most similar image among a great amount of images, avoiding the trouble to have to deal with an expensive computation time while using brute-force.