Locality Sensitive Hashing

Rebecca C. Steorts

October 17, 2017

1 Locality Sensitive Hashing (LSH)

A *hash function* maps objects to integers such that dissimilar objects are mapped far apart. LSH uses special hash functions to ensure that similar objects are put close to each other (with high probability). By applying several locality-sensitive hashes to a record, one goes from a high-dimensional object to a low-dimensional signature, with a guarantee that similar records have nearby signatures. The low-dimensional signature vectors can then be divided into bins or blocks, with a high probability that all records mapped to the same bin are similar, and a hope that not too many similar records fall into different bins. If the number of records per bin is small, we can treat the bins as blocks for purposes of record linkage. The number of records per bin depends on the exact binning procedure and its tuning parameters (Kim and Lee 2010; Rajaraman and Ullman 2012; Christen 2012). Unlike conventional blocking, LSH uses all the fields of a record and can be adjusted to ensure that blocks are manageably small.[1] By design, records falling within the same block are similar to each other, so only linking records within blocks can lead to dramatic speed-ups while imposing only a small cost in false negative errors. (One does fewer comparisons, and those comparisons are more likely to lead to links.)

While these are all desirable features, LSH is in several ways not yet fully developed as a blocking technique for entity resolution. One is that "similarity" must be made computationally precise, and not every similarity measure is preserved by a (known) family of hash functions. Moreover, different similarity measures may be appropriate for different kinds of data in different applications. Second, entity resolution should come not from ranking similarity scores (as in Liang et al. 2014), but from a statistical

[1]The last point needs some care, since simulation studies show that *sometimes* bins can contain many records, leading to minimal computational savings.

model. We develop a variety of LSH blocking algorithms which are designed to work well with real-world examples of entity resolution and to meet the needs of my statistical models.

1.1 Minwise hashing

Minwise hashing is a procedure for comparing the similarity of two sets (records in this context can be thought of as sets). For a large dataset like the Syrian one we are considering, minwise hashing is optimal because of its speed in subdividing records into blocks. The implementation of minwise hashing has a number of steps.

First

We shingle each record. Shingling is a way of splitting apart a record into smaller subsets, at some specified splitting point. For example, if we have the record [“Peter”, “Pittsburgh”], and we want to use a shingling of size k = 3, we form the shingles [“Pet”, “erP”, “itt”, “sbu”, “rgh”]. To do this, we combine the entire record into a single string, and then subdivide it into parts of three.

Shingling is done to each record in the dataset, which forms a large set of shingles. We also want to keep track of which records are associated with which shingles. So if we have a second record, [“Steve”, “Pittsburgh”], then we know that the shingles “itt”, “sbu”, and “argh” are in common between the two records.

Second

We form a characteristic matrix from the shingles. A characteristic matrix is an indicator matrix, where the columns of the matrix correspond to records in the dataset (so there is a column for each record) and the row to the shingles formed from the records (so there is a row for each shingle). The elements of the matrix are a binary (1, 0) decision, where a 1 is present if the record contains the corresponding shingle, and a 0 if not. The characteristic matrix is very sparse (i.e., it contains mainly 0s), as most records do not contain the majority of all possible shingles.

Table 1 is an example of a characteristic matrix, with four records(sets) and five shingles(elements).

Third

We permute the rows of the characteristic matrix to form a permuted

Element 1-5	S_1	S_2	S_3	S_4
1	0	0	1	1
2	1	0	0	1
3	1	1	0	0
4	0	0	0	1
5	1	0	0	1

Table 1: A sample characteristic matrix, with four records in columns and five possible shingles in rows. As the number of shingles grows, the matrix becomes increasingly sparse, as most records do not contain most shingles.

matrix. If we have five rows, 1-5, then possible permutations of the rows are 12345, 15234, and 54321 (there are many others). The permuted matrix, then, is simply a reordering of the original characteristic matrix, with the rows swapped in some arrangement. Figure 1.1 shows the characteristic matrix converted to a permuted matrix by a given permutation.

We repeat the permutation step for several iterations to obtain multiple permuted matrices.

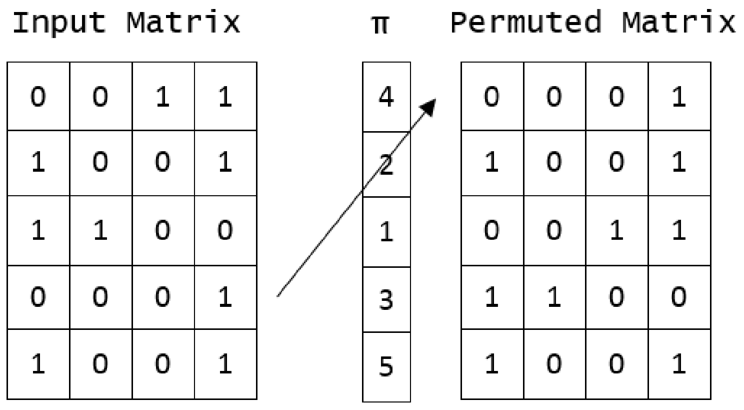

Figure 1: Permuted matrix from the characteristic one. The π vector is the specified permutation.

In practice, the implementation of minhashing is slightly different from these permutations, as permuting the matrices a large number of times is computationally-prohibitive. To make up for this, a series of hash functions is generated, where the hash function assign values to each row, in exactly the same way as the permutations do. The hash function, $(r + 1)\%5$, for example, takes in a row index (1-5), and assigns the new indices as the permuted matrix. This results in the same outcomes as the one above. An example of these hash functions with the characteristic matrix is given in table 2.

Row	S_1	S_2	S_3	S_4	(r+1) % 5	(3r+2) % 5
0	0	0	1	1	1	2
1	1	0	0	1	2	0
2	1	1	0	0	3	3
3	0	0	0	1	4	1
4	1	0	0	1	0	4

Table 2: Characteristic matrix with four sets, two hash functions, and a generated binary matrix. The hash values are found by inputting the corresponding row index to the hash function.

Fourth

We compute the signature matrix. The signature matrix is a hashing of values from the permuted one. The signature has a row for the number of permutations calculated, and a column corresponding with the columns of the permuted matrix. We iterate over each column of the permuted matrix, and populate the signature matrix, row-wise, with the row index from the first 1 value found in the column. The row index inputted to the signature matrix is the new row index the row was associated with, rather than the original index value. The signature matrix for table 1 is in table 3.

	S_1	S_2	S_3	S_4
h_1	2	4	3	1

Table 3: Signature matrix formed from table 1. The value for each element is the row index in which the first 1 is found in the permuted matrix.

Fifth

Based on the results of the signature matrix, we observe the pairwise similarity of two records. For the number of hash functions, there is an interesting relationship that we use between the columns for any given set of the signature matrix and a Jaccard Similarity measure.

The Jaccard Similarity, which is a measure of similarity, is fundamental in this approach. For two sets S and T, the Jaccard Similarity is given by

$$\frac{|S \cap T|}{|S \cup T|}$$

which is the intersection of the two sets over their union. It provides a sense of how much two records agree in their fields, over their total number of variables.

The relationship between the random permutations of the characteristic matrix and the Jaccard Similarity is:

$$Pr\{min[h(A)] = min[h(B)]\} = \frac{|A \cap B|}{|A \cup B|}$$

The equation means that the probability that the minimum values of the given hash function, in this case h, is the same for sets A and B is equivalent to the Jaccard Similarity, especially as the number of record comparisons increases. The output of this formula, in terms of the signature matrix, can be seen in figure 1.1.

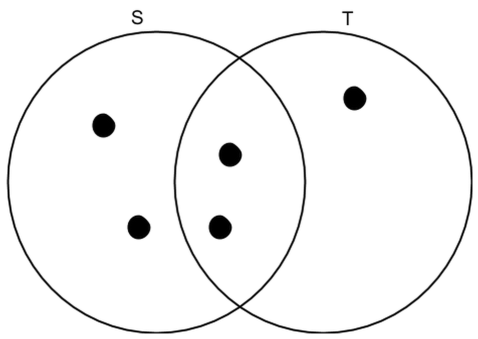

Figure 2: Two sets S and T with Jaccard similarity 2/5. The two sets share two elements in common, and there are five elements in total.

We use this relationship to calculate the similarity between any two records. We look down each column, and compare it to any other column: the number of agreements over the total number of combinations is equal to Jaccard measure.

Of course, in practice, we would need to repeat the above for many permutations π such that we avoid collisions, or mapping the different records into the same bin.

Notice that we have walked through a very elementary view of locality sensitive hashing, and we still have the limitation that we are having to do all-to-all comparisons. How do we avoid this?

2 LSH for Minhash Signatures

See Section 3.4.1 in Mining for Massive Datasets and work through the exercises (some of these we did in class). See Example 3.11 and Exercise 3.4.1 and 3.4.2.

3 Further reading

For further reading about hashing and it's use in practice, please refer to https://arxiv.org/abs/1710.02690.

4 Overview of Locality Sensitive Hashing (LSH) Algorithm

Below we outline the LSH algorithm (for all-to-all comparisons) and also for filtering out documents that are not very similar.

1. Construct shingles of all documents in your corpus.
2. Hash all of your shingled documents.
3. Compute pairwise Jaccard similarity coefficients for all documents.
    (a) To do this in a computationally more efficient way, use the characteristic matrix and a random permutation.
    (b) Then create the signature matrix by using the minhash. Repeat this process using many random permutations in order to avoid collisions. This will increase the size of your signature matrix.

To avoid performing all-to-all comparisons as in the above situation, we compute the Jaccard similarity only for candidate pairs using b bands and r rows of the signature matrix, which provide a threshold $t = (1/b)^{1/r}$ using the steps above but now using these extra conditions of filtering out documents that are unlikely to be the same.

References

Christen, P. (2012). *Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection.* Springer.

Kim, H.-s. and Lee, D. (2010). Harra: fast iterative hashed record linkage for large-scale data collections. In *Proceedings of the 13th International Conference on Extending Database Technology.* ACM, 525–536.

Liang, H., Wang, Y., Christen, P. and Gayler, R. (2014). Noise-tolerant approximate blocking for dynamic real-time entity resolution. In *Eighteenth Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD14)* (Z.-H. Zhou, A. L. P. Chen, V. S. Tseng and B. H. Tu, eds.). Springer, New York, NY, forthcoming.

Rajaraman, A. and Ullman, J. D. (2012). *Mining of massive datasets.* Cambridge University Press.