Locality Sensitive Hashing

Martijn J. Post & Vincent P. Post

Abstract. Locality sensitive hashing is a fast and memory efficient algorithm to obtain an estimate of the amount of similar items in a dataset compared to an exhaustive search. In this report we discuss our implementation of the locality sensitive hashing algorithm combined with techniques called shingling, minhashing and banding. We test the algorithm on a dataset of 103 700 users who rated in total 17 770 movies, each user rated at least 300 movies. We search for the amount of user pairs that rated movies with a similarity larger than 0.5. The algorithm returns over five runs a median value of 417 user pairs that match the similarity condition within a median time of 403 seconds on an Intel Core i5-6300U CPU @2.4 GHz with 8 GB of RAM. The result is found to be off from the correct value of 1205 user pairs. However in comparison to an exhaustive search, which would take 106 days, the algorithm is significantly faster.

1. Introduction

Exhaustive searching in a large data set for items that are similar to some degree can take a long time on a standard consumer computer. As a consequence clever algorithms are developed that give an estimate of the amount of items that are similar to some degree in a reasonable time span and within the working memory of a consumer computer. We create such a clever algorithm by combining four ideas: shingling, minhashing and locality sensitive hashing (LSH) with banding to obtain user pairs that rated a set of movies with a similarity larger than 0.5. The implementation of these techniques is based on the implementation/theory shown in chapter 3 of the book "Mining of Massive Datasets" [2].

We obtained a data set which contains 103 703 users that rated in total 17 700 movies of which every user rated at least 300 movies. The dataset originates from the original Netflix Challenge [1], but the users that rated less than 300 or more than 3000 movies are left out. The first step in handling the dataset is to apply shingling to build a matrix of the data set, hence putting the data set in a form to calculate the similarity. Shingling in this case simply means that the movies are the shingles and if a user rated that movie we obtain a 1 in a matrix and 0 if the user did not rate it. As a consequence the matrix is a sparse matrix.

2. Methods

To calculate the similarity of user pairs that rated movies from the sparse matrix we use the Jaccard similarity. The Jaccard similarity between two sets is defined as follow $Jsim(u1, u2) = |u1 \cap u2|/|u1 \cup u2|$. In which $u1$ is one user and $u2$ is the other user. We will force $u1$ to be smaller than $u2$ because if $u1 = 1, u2 = 3$ then $u1 = 3, u2 = 1$ is the same pair and we will be double counting. Before we discuss our implementation of minhashing and LSH we calculate how much time is needed to do an exhaustive search over the whole data set to obtain all user pairs that have a Jaccard similarity

larger than 0.5. The total number of pairs to calculate the Jaccard similarity of is: $\binom{103\,703}{2} = 5\,377\,104\,253$ pairs! To get an estimate of how much time the calculation would cost on our computer, an Intel Core i5-6300U CPU @2.4 GHz with 8 GB of RAM, we wrote a python program to calculate the Jaccard similarity of 1000 user pairs. The median run time over 5 runs was approximately 1.701 seconds. If we would calculate the time it would take to calculate the similarity over all user pairs it would take: $5\,377\,104\,253/1000 \cdot 1.701 \approx 9.15 \cdot 10^6$ seconds which is approximately 106 days. In the calculation we neglected the fact that in our calculation we looped over all user pairs we created, which would be far to many user pairs to hold in our RAM. For this method to work we think it would need to write and call upon the hard disk multiple times which would increase the run time even more.

A run-time of 106 days is of course not efficient for calculating the similarities of the users because by then the result may not be needed anymore. An example is if we would need to match fingerprints in a large data set to help with solving a crime, we would like to obtain results within a few minutes and not at least a few weeks. Therefore we employ the minhashing and LSH with banding techniques to give an estimate of the user pairs that have a similarity larger than 0.5. By comparison if we would want to do this within the RAM and within a few minutes this will come at the cost of not finding all the user pairs that match the similarity condition.

We start with the minhashing technique, which reduces the large sparse matrix to a so-called signature matrix. The signature matrix is created by randomly permuting the row order of the sparse matrix and denoting the row position of the first 1 for each user in the signature matrix. The users form the columns of the matrix, the rows are the amount of random row permutations applied and the values in the matrix are thus the position of the first 1 for each user. With the obtained signature matrix we can calculate the similarity of users. This calculation is done by summing over all the values that are the same for the two users and dividing by the amount of rows of the signature matrix. We set the amount of random row permutations equal to 100 because it is found from the book [2] and from small experiments that the similarity values obtained are close to the corresponding Jaccard similarities. The max deviation between similarities seen in our runs was 0.08.

The next step is to apply LSH with banding. With this smaller signature matrix we still need to evaluate each user pair to check for a similarity larger than 0.5, with the 103 703 users this still costs quite some time. Therefore we use LSH with banding, it splits the signature matrix into bands in which each bands contains all user columns but only a few rows. The reason for the bands is that when two users are similar there will be a high probability that in at least one band they will have exactly the same column values. This smaller column in a band will be referred to as a vector. Furthermore if user vectors are the same we will hash them to the same bucket. As a consequence we obtain per band a bucket array and if a bucket contains more than one entry the user entries form candidate pair. Candidate pairs are often similar to each other. The probability that two users become a candidate pair is equal to: $1-(1-s^r)^b$ with s being

the Jaccard similarity value, r the amount of rows and b being the amount of bands [2]. The amount of rows is set by the amount of bands and the amount of row permutations made for the signature matrix, the corresponding equation is $r \cdot b = p$ with p the amount of row permutations. For our data set we want the probability that two users become a candidate pair at $s = 0.5$ to be as high as possible. However a problem arises because at different values of b or p we always obtain a 'tail' of non-zero probabilities for lower s values, we want this probability to be 0. To push these probabilities down to 0 we can increase p but increasing p significantly increases the run time and the memory usage of the algorithm. Therefore a decision needs to be made on what the user wants, our choose is shown in the results section.

Our implementation of LSH with banding records all the the buckets per band that have more than one user in it and append it to a larger bucket list. The final bucket list thus only contains candidate pairs. As an example the bucket list looks as follow: $[(1, 2, 4, 5), (6, 9), (8, 10, 23)]$ however this list can contain exact duplicate tuples. Therefore in the next step we remove the exact duplicate tuples. The last step is iterating over the bucket list and creating per bucket a python set of each combination in a tuple. When a combination(a user pair) is not already in a so-called unique set, the similarity is calculated with the signature matrix. If the similarity is larger than 0.5 it is put into the unique python set. It is important to note that the similarity calculation is done over the signature matrix such that the run time is lower. In the end we determine over the found user pairs, in the unique python set, the Jaccard similarity, when this similarity is larger than 0.5 it is written to a txt file. Also, during this last step it is checked if user 1 is smaller than user 2 if this is not the case the user 2 becomes user 1 and vice versa as long as no duplicates begin to form.

3. Results

We call our similarity algorithm using the command: "python lsh.py <Random Seed> <full path to dataset and file name>". The first argument is the random seed which can be any integer and is used for creating a random permutation for use with minhashing. The second argument is the path of the dataset and file name "user_movie.npy". In Table 1 the results of our similarity algorithm over five different runs with different random number generator seeds are shown.

For our experiment we set the amount of row permutations equal to 100 and the number of bands equal to 20 which gives as the number of rows 5. As can be seen in the table is that the median over 5 runs the found pairs is equal to 417. However the dataset contains 1205 users pairs with a similarity larger than 0.5. On the contrary that we miss quite some user pairs we obtain a short run time of which the median over 5 runs is equal to 403 seconds or just under 7 minutes compared to approximately 106 days.

To raise the amount of similar pairs found, we can increase the amount of bands

Table 1: Results over five runs from the locality sensitive hashing algorithm with banding. Here we have applied 100 random row permutations, and set the number of bands equal to 20 from which we obtain 5 rows per band. Shown are the random number generator seeds used, the amount of user pairs found and the run time in seconds of each corresponding run. Also, the median over the runs is shown.

	seed	user pairs	run time (seconds)
run 1	42	404	403
run 2	69	497	630
run 3	169	417	310
run 4	54321	586	431
run 5	13	349	308
median over runs	-	417	403

in our LSH algorithm from 20 to 25 with 100 permutations which would change the probability to find a candidate pair with a similarity of 0.5 from 47% to 80%, calculated with the function shown in methods. We did try using 25 bands in practice and while we did find more similar pairs, during one run 600 pairs, the run time of the algorithm would rise significantly (at least more than 20 minutes) and the required memory in RAM would also rise. The required memory would sometimes go over the 8 GB possible in the computer, making the algorithm crash or become very slow due to writing data to the hard disk. Therefore we decided against using 25 bands because we did not want to risk the required time exceeding 30 minutes for an unlucky random seed. Changing the total amount of permutations would also increase the probability of finding more similar user pairs but would bring the same downside of requiring significantly more time and memory to find the candidate pairs.

4. Conclusion

In conclusion we see that using the similarity algorithm which uses shingling, minhashing and LSH with banding techniques for determining candidate pairs with a Jaccard similarity larger than 0.5 does not find all the user pairs that match this condition. However it is very efficient in the amount of memory used and time needed to find a part of the total amount of similar user pairs that match this condition. When the algorithm is compared to an exhaustive search which would take approximately 106 days to find the approximate 1205 user pairs, finding 417 user pairs in 403 seconds and a lower amount of needed memory might be a better alternative for quickly determining the user pairs that match a similarity condition.

[1] Netflix challenge. http://www.netflixprize.com/.

[2] J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of massive datasets.* Cambridge University Press, 2014.