CS422 Project 2 - Approximate Near-Neighbor Queries using Locality Sensitive Hashing

In this report, I will present my code for Project 2, together with the results I obtained. The project's goal was to implement, through the Apache Spark framework, different approximations of the Near-Neighbor algorithm, using Locality Sensitive Hashing.

Exact Near Neighbor

As a baseline for the following tasks, I have to implement an Exact Near Neighbor algorithm, working with Spark RDDs as input. In practice, given an RDD of queries, I have to find all the entries in the data RDD that match a specific criterion for each query. In this case, the criterion is the Jaccard similarity between the set of query keywords and the set of keywords of the data entry, which has to be above a certain threshold parameter.
So, my implementation is pretty straightforward: a cartesian product between queries and data, then I compute the Jaccard similarity for each entry of the result, and I filter for the rows that match the criteria. After filtering, to have an RDD structured like [(query, List(Results)], I apply a ReduceByKey transformation.

Of course, the cartesian product, together with the computation of the similarity for each row, makes this implementation, although very accurate, too expensive for large datasets, as we will see in the *Result* section.

Base construction and Min Hash computation

As said during the project presentation, an idea to reduce the computation needed for this task is to map the set of keywords of each query, and each data entry (a movie, in our case), to a single integer. The idea is relatively straightforward: all the unique keywords are mapped to an integer through a Hash function, and then we represent each movie with the minimum among all the integers obtained from its keywords.
Of course, we lose some information in the process, since each movie is only associated with a single keyword. On the other hand, we avoid the previously explained cartesian product and similarity computation, as now we only have to group the movies by their min hash (obtaining the so-called buckets), and then perform a join between the queries and these buckets.

Balanced construction

Here we start thinking more about how Spark partitions the data across all the workers, and try to optimize this process: with the base construction presented before, the buckets are equally distributed among the workers, but this is not always desirable: if one partition receives many more queries than the others, it will become a straggler, making the execution of the entire job slower. In this construction, I partition the buckets according to the number of queries that end up in each of them, balancing this number with respect to the workers.

First of all, in the computeMinHashHistogram I count, for each min hash values in the query file, how many queries I have, and I sort the queries according to their min hash value. So, I end up with an array (not an RDD) like [(minHash, numberOfQueries)], sorted by minHash.

Then, in computePartitions I compute the boundaries of each partition, with this logic: first of all, given the number of partitions I want to obtain, I compute the number of queries for each partition (ceil of totalNumberOfQueries/numberOfPartitions). Then, I loop on the histogram. For each element, I sum the number of queries to an external variable. If the sum gets bigger than the number of queries for each partition, the related histogram element represents a boundary. I put the sum again to zero and continue. In the end, I obtain an array structured like:
[minHashPartition1 +1, minHashPartition2 +1…]. Note the +1 that I need for my logic (please take a look at the code if you do not understand my choice).

Finally, I assign each bucket to its partition, according to its min hash value, grouping by the partition number. This means that in the end, I end up with an RDD of (at most) P elements, where P is the number of desired partitions. Each entry will contain the partition ID and the Array of buckets.
I do the same with the queries, and then I join the two RDDs on partition ID. This means that now I have all the queries and buckets for each partition inside a single entry of this final RDD.
Then, for each row of this RDD, for each query, I find its bucket.

Three considerations must be made here:

- Since I do not split a bucket, if I have bucket A, with 100.000 queries, and bucket B, with 10 queries, with P>1, I will always have two partitions, one with bucket A and one with bucket B. Of course, the first partition will be a straggler.
- My computation of the histogram is a complete approximation: think about the following example: I have three buckets, and the following histogram [(id1, 1), (id2, 3), (id3, 1)], with id1 < id2 < id3. Now, the best split, in two partitions, would be P1 = [(id1, 1), (id3, 1)], P2 = [(id2, 3)]. But with my solution I end up with P1 = [(id1, 1), (id2, 3)] and P2 = [(id3, 1)]. Now, with a high number of buckets and queries, this approximation seems to work well, but there might be some cases for which a better partitioning could lead to way faster results.
- All the operations I described take time, and thus, if the number of queries is not really high, this algorithm performs worse than the unbalanced one.

Talking about performances, when using the *Corpus 10* and *Queries 10-2* files (taking all the queries), with 8 partitions, the number of queries for each partition is:

- 5100
- 5100
- 5100
- 5100
- 5100
- 5020
- 5100
- 5100

Now, when looking at *Queries 10-2-skew*, the situation I described above, in consideration number 1, happens :

- 4354
- 8579
- 4365
- 4361
- 5853
- 2946
- 4356

Note that, even if P=8, I end up with 7 partitions in this second case. This is expected with the algorithm I described, as I effectively split the histogram only when the cumulative number of queries surpasses the computed number of queries for partition, and, although this number is a function of P, there is no guarantee that the number of splits is the same as the number of partitions.

Broadcast construction

In this construction, we try to reduce the shuffling operations, broadcasting all the buckets to every partition. If you think about the base construction, indeed, the join operation requires a shuffle, because queries in

partition X might refer to a bucket that is in partition Y. Of course, the higher the number of queries and buckets, the higher the amount of data that is transferred across the workers.
Broadcasting all the buckets to each worker means that the shuffle phase is avoided. On the other hand, however, it also means that each worker has to keep in memory all the buckets as a broadcast variable and that the data must effectively be broadcasted to each executor, which could cause problems if its size is considerably high.
In my implementation I simply collect the buckets, once created, on the driver, and then I send them, through a Spark broadcast object, to each worker. Then, with a map operation, I compute the result.

And and Or constructions

To achieve better performances in terms of precision or recall, we can associate to each movie more than one integer, repeating the hashing operation many times, with different seeds. So the logic here is: first of all, create N constructions with N different seeds. They will give different results when evaluated.
Then, for the And construction, for each query, pick as the total result the intersection of the results given by the various constructions. This increases the precision with respect to the ground truth, which is represented by the ExactNN, as we will have many fewer false positives with this method. Concerning the OR construction, the total result is the union of the results given by the construction. This, of course, increases the recall, as we will reduce the number of false negatives.

Results

In this section, I will make some considerations about the results of this project. Note: all the tests have been executed on my local machine, a Macbook Pro 13" with Apple M1 SoC. The seeds are: 42 for the base constructions, 42-43-44-45 for the AND construction and 42-43-44-45-46 for the OR construction.

Firstly, I report precision and recall of the constructions (Base, Balanced, Broadcast), with respect to the ground truth given by the Exact NN. For this test, I will use *Corpus 10* and various query files, with 5% of the queries taken into account, as it becomes impractical to compute the ExactNN on a more significant number of queries. The ExactNN threshold is set to 0.3. I also compute the Jaccard distance among the results (*lower is better*). This is how it works: for each query, I compute the Jaccard Distance with each movie retrieved for that query, and I average the result. Once I have a result for each query, I average again.

Please note that each construction among Base, Balanced and Broadcast presents the exact same results, as it is not the logic that changes among them, but just how we execute that logic.

Query File	**Precision**	**Recall**	**Jaccard Distance**	
Queries-10-2	0.38	0.81	ExactNN	Construction
			0.21	0.64
Queries-1-10	0.36	0.79	0.21	0.65
Queries-1-2	0.36	0.80	0.21	0.66

As expected, when looking at the Jaccard Distance, the ExactNN algorithm performs way better than our approximations. However, as said, the ExactNN execution time is simply prohibitive when working with large datasets. Now we repeat the same test, but with AND and OR constructions, to see how the three reported metrics change. We will use four balanced constructions in AND and five in OR (as in the functions *construction1* and *construction2*). We do not combine the two logical operations.

AND CONSTRUCTION

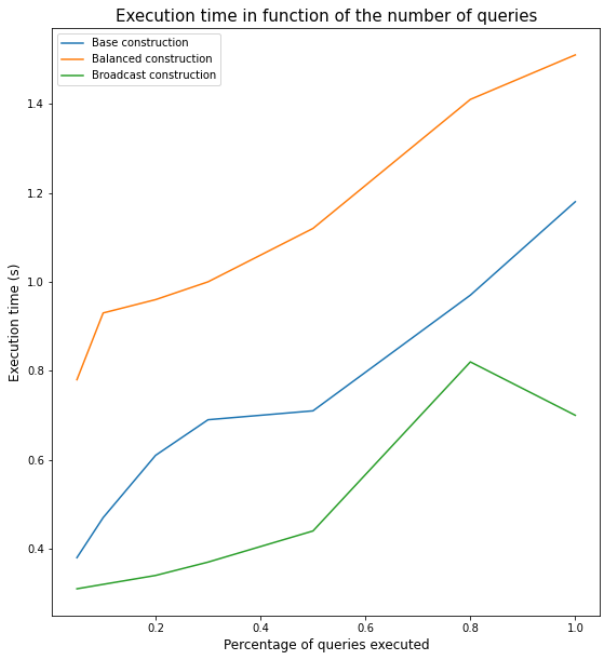
Query File	Precision	Recall	Jaccard Distance	
Queries-10-2	0.99	0.66	ExactNN	Construction
			0.21	0.03
Queries-1-10	0.99	0.67	0.21	0.04
Queries-1-2	0.98	0.68	0.21	0.03

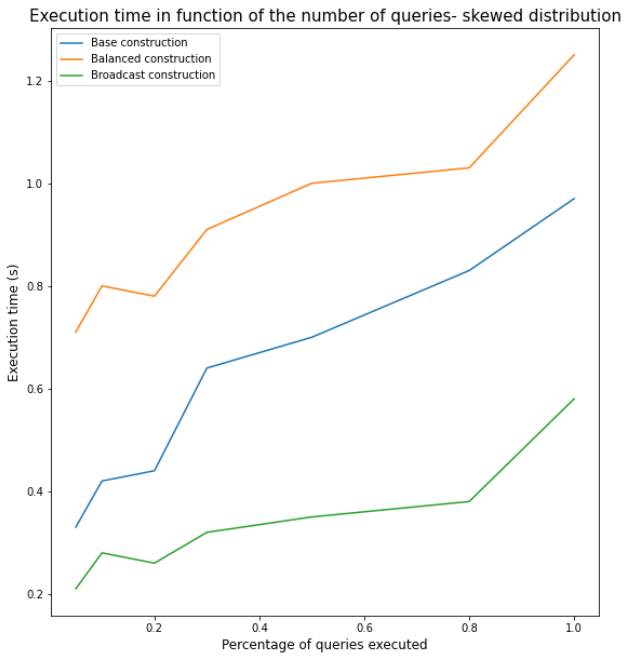
OR CONSTRUCTION

Query File	Precision	Recall	Jaccard Distance	
Queries-10-2	0.22	0.96	ExactNN	Construction
			0.21	0.77
Queries-1-10	0.23	0.96	0.21	0.77
Queries-1-2	0.20	0.95	0.21	0.79

We notice a sharp increase in precision when evaluating the AND constructions and in recall when evaluating the OR construction. Moreover, the Jaccard distance in the AND construction is close to 0, as expected.

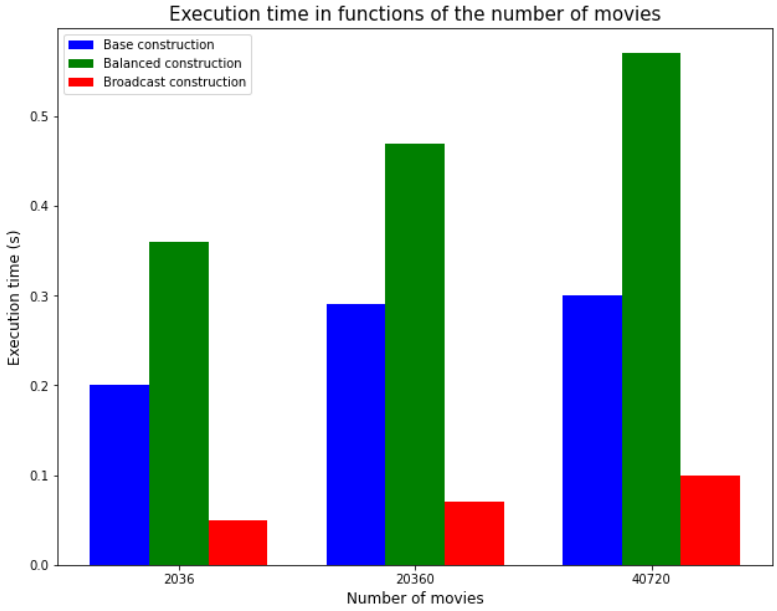
For the third test, I compare the execution times of the three constructions in function of the number of queries. The setting is the following: I use the *Corpus-20* file, together with *Queries-20-10.* This file has 407200 queries, and I increase the percentage of queries considered to create the following plot. Then, I repeat the same test with *Queries-20-10-skew* to see if something changes.





We spot some interesting behaviors here. Firstly, the number of queries is still not enough for the Balanced construction to perform better than the Base one. We need a more significant number to really see the power of this construction.
Secondly, we notice that the Broadcast trick is always faster than the two other constructions.

We repeat the same analysis, but instead of varying the number of queries, we analyze how the execution time changes when considering a different number of movies. In particular, we use *Queries-1-10* (100% of the queries) as query file and *Corpus-1, Corpus-10* and *Corpus-20* as movie files.



We can notice that the broadcast construction scales well with respect to the number of movies. However, my opinion is that if we raise the number of movies and thus the size of each bucket higher than a certain threshold, we could see a performance degradation due to the memory bandwidth necessary to broadcast the buckets.

In conclusion, I would say that if the number of movies and queries is minimal, there is no reason to not use an exact algorithm, like the one implemented in the ExactNN class. If the number of movies /queries increases, as long as the movies are not enough to create problems with broadcasted, the Broadcast construction is the right choice.

I am also sure that we did not exploit the advantages offered by the Balanced construction, which could perform better than the others with an even higher number of queries.

Then, with respect to the accuracy, it really depends on what we ask: if we do not care about recall but we want to highly reduce the number of false positives, we can use an AND construction. If, vice versa, we care more about the recall and less about the precision, we will use an OR construction. We can also think of mixing the two to achieve better performances in general in terms of the two metrics.

Giorgio Mannarini