Minhash/LSH and DuckDB

Massimini Angelo   Pugnaloni Francesco

Contents

1 Locality Sensitive Hashing

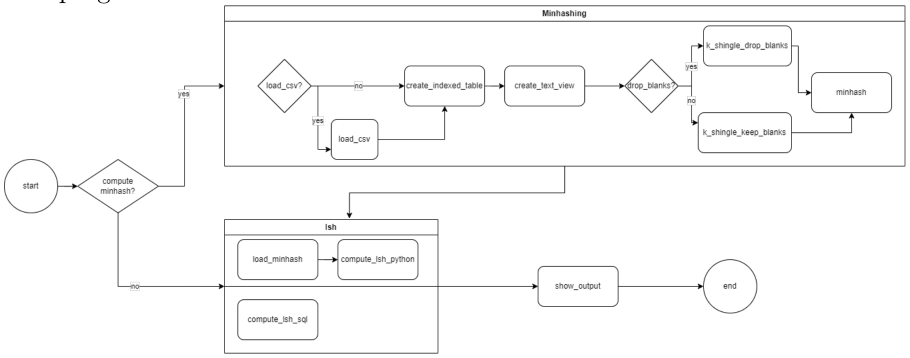
To identify similar items is useful to use Locality Sensitive Hashing(LSH), a technique developed in steps.
The first step, called shingling, aim to convert input documents into shingles, that are substrings. To start this phase we need to define k-shingle, the length of substrings. This choice depends on the context, in fact if k is too small is more likely to find a shingle in a document, while if the lenght is too big it is harder. Then we can associate each document with the shingles that belong to it, in a table, using boolean values. However is expensive to compare sets of a lot of shingles. So we need to use hashing techniques to create signatures and minimize the comparisons and in particular we use minhash.
Minhash creates these signatures using permutations of the columns(shingles) of the shingle table. The value of each singature is the number of the first column, in the permutated order, in which the row has value true. Minhash is a good techninque because it doesn't change Jaccard similarity between the items so at the end of the LSH we can compare minhash results with the LSH results to delete false positives. However, the use of minhash only is not enough, because we have a lot of signatures to compare; so we need to use locality sensitive hashing. LSH expects the joint use of buckets, containers in which to place similar candidate documents, and banding, because to find the similarities we divide singatures in short parts so that we can use them to quickly compare our elements; two elements will be similar candidates if they hash in the same bucket in a band. At the end of LSH we will have buckets containing sets of similar candidate documents. Now we can proceed with the elimination of false positives computing the actual jaccard similarity using the minash table.

2 DuckDB

DuckDB is an open-source, relational database management system with high-performance analytical capabilities designed to be fast, reliable, easy to use and to be embedded into other applications; indeed we can include it as a library in our code. It is a columnar database, that is organised by field, keeping all the data associated with a field next to each other. This is useful for Online Analytical Processing system (OLAP) workload because most OLAP queries typically access a large number of rows but need to use only a small subset of the columns. It uses vectorized data processing which helps to develop faster analytical engines by making efficient utilisation of CPU cache. It is ACID compliant, an important feature for relational databases that need to ensure data consistency and integrity; it guarantees that the database will be in a consistent state, regardless of any failure that may occur during a transaction, and that once a transaction has been committed, its changes will persist in the database. DuckDB can be used in a wide range of applications where low-latency and high-performance data processing is needed, like machine learning or edge computing.

3 Overview

The project's objective was to produce a python implementation of the minhash lsh algorithm to detect similar tuples (in terms of jaccard similarity) inside a textual dataset provided via duckdb or csv file. Furthermore directly processing data inside duckdb using sql whenever possible was suggested in order to test duckdb's optimizations.
The program follows the code flow below:



The execution is divided in two major phases that are minhash phase and lsh phase; it is notable to say that executing both every time is not mandatory, in fact during the execution a "checkpoint" is written inside the duckdb file.

Let's now explain the two phases more in details:

Minhash It is the first one and it aims to create a minhashed representation for every document in the dataset using an arbitrary number of textual columns; its main steps are the following and they aren't all mandatory:

1. Duckdb file creation from a csv file: if there isn't an existing .duckdb file it is possible to create a new one using the load_csv function, this new file will contain only one table that is the relational representation of what is provided

2. Indexed_table creation: a new table is created from the old one adding a new column that is an increasing index, it is necessary because we will use it to identify the tuples in the following steps in a standardized way

3. Text_view creation: basically a view over the indexed table is created, it will contain only two columns: the identifier and the concatenation of the textual column to use to compute the minhash

4. Shingling: in this step the shingling operation is performed: the created view is loaded in a numpy array and it is used to create a compressed shingle matrix in the form of a list, where every element is associated to a list of integers that represents the positions in the shingle matrix where a 1 occurs . This operation can create the shingles in two different ways depending on the parameters: dropping whitespaces from the text

or keeping them. Furthermore it is possible to set a threshold to limit the number of shingles generated

5. Minhashing: finally the minhashing operation is performed, the output of this phase will be a dictionary containing two entries: the identifiers of the tuples as a 1D numpy array and another 2D numpy array with shape number_of_documents X 128, where every row represents the minhash of the corresponding document. It is notable that the permutations are computed using the universal hashing function and it is possible to set a threshold to stop the minhashing operation before it tries all the possible permutations to speed up the process.

6. Minhashes writing: the output dictionary is persisted inside the duckdb database as a table having a row for each document and 129 columns, the first one is the index of the tuple, the other 128 are its minhash. This operation is vital because minhashing is a very expensive operation and avoiding its recomputation every time we need to modify a parameter in the lsh funciton saves us a crazy lot of time

LSH In this second phase the lsh operation can be performed in two different ways (or in both just to run tests):

1. numpy-python: the minhash table is ridden inside a dictionary with the previously described structure and the operations are performed using numpy's functions to work with arrays

2. SQL-duckdb: in this case a read is not necessary and all the operations are performed using just pure SQL

Regardless of the specific implementation both functions perform almost the same operations that are the creation of candidates pairs using the bucket approach (the function used to fill the buckets is the sum) with the bands technique, and the false positives elimination computing the jaccard similarity. Then the pairs found are saved inside the database in a new table that contains couples of indexes that are the ones created during the initial stages of the execution and can be joined with the indexed table to retrive the content of the similar documents.

Program's usage Everything revolves around a function that encapsulate all the others, it is called "lsh", there are 3 mandatory parameters (database name, table name and column list) and 12 optional. Anyway all the parameters are exhaustively described inside the notebook with the code.

4 Tests performed

Dataset The used dataset can be found here https://www.kaggle.com/datasets/jdobrow/57000-books-with-metadata-and-blurbs. It is a free dataset from the platform "kaggle" and contains data about 57000 books. The attributes on which we focused our analysis are the title and the blurb (a small summary of the book).

Test performed The aim of these tests is to compare the execution times of the 2 versions of lsh as the number of records varies, fixed all the other parameters. It is notable that the value of k (the shingle length) is fixed to 5, this is not a random choice because blurbs are similar in length to mails, so a huge k is not needed to obtain a good result. Other important parameters are the number of bands (fixed to 16) and the similarity threshold (0.7). The followed workflow is simple: using a for loop the function is executed six times processing and increasing number of rows in every iteration, in the end the results are saved in a table like data structure.

From the tests executed it appears that the python version is faster than the other, the result obtained are the following:

	Python	SQL
100 rows	99.96 ms	422.06 ms
500 rows	141.32 ms	530.91 ms
1000 rows	199.98 ms	701.22 ms
5000 rows	945.99 ms	3748.53 ms
10000 rows	2011.01 ms	11961.82 ms
50000 rows	31106 ms	336867 ms

5 Conclusion

Features We found duckdb a usefull tool when it comes to local relational data processing (because it is fast to use and only needs a single file) and especially appreciated the possibility to write sql queries using both relatioal tables and pandas data frames together. We also liked the possibility to convert a relation in dictionary of numpy array to speedup the ORM (Object Relational Mapping) work.

Performances Like we showed in the previous section from our tests the python implementation performed better than the SQL one. The possible reasons for this can be various. For sure using the numpy library boosted python performances because it substantially uses C arrays. Another possible reason is that the lsh task doesn't exploit the optimizations provided by duckdb at its best because it needs to read almost all the columns of the minhashes table.