UNIVERSITÀ DEGLI STUDI DI MILANO

Algorithms for Massive Data Analysis Paper
Finding similar items project
Privacy, Data Protection, and Massive Data Analysis in emerging scenarios

Michela Mazzaglia
michela.mazzaglia@studenti.unimi.it

June 2024

Abstract

This project was implemented with the aim of building a detector to find similar items based on the rows from the *job_summary* column of the csv file, which contain descriptions of job offers posted on the social network *LinkedIn*. The Map Reduce architecture was used to compute LSH (Locality-sensitive Hashing) using a Minhash signature on Apache Spark.

Contents

1 Dataset and Preprocessing

The project aims to construct a detector to find similar items. The data set was downloaded through the Kaggle API directly on the Google Colab platform. It collects the links and descriptions of job offers posted on *LinkedIn*. After having installed all the necessary packages, imported the libraries and initialised the Spark session, there rised some problems in reading the csv file on Spark. However, this issue was handled with two solutions. On one hand, a schema was defined in order to let Spark understand exactly the type of object of both the job link and job summary columns. Moreover, Spark was instructed to handle special characters, quotes and multiline fields, to avoid to mix the content inside the two columns which happened in the first trial. Since this first solution did not solve the issue overall, the second one implemented included the handling of null values, the substituing of the special character `\n` with spaces and by splitting all the words inside the descriptions creating an array of words. This last operation made possible to deal with those rows that were incorrect and to maintain job descriptions of more than 4 words. This is an example of the Spark dataframe:

```
+--------------------+------+
|           job_words|doc_id|
+--------------------+------+
|[Rock, N, Roll, S...|     0|
|[Schedule, :, PRN...|     1|
|["Description, In...|     2|
|[, we, encourage,...|     3|
|[Commercial, acco...|     4|
|[Address:, USA-CT...|     5|
|[Description, Our...|     6|
|[Company, Descrip...|     7|
|[An, exciting, op...|     8|
|[Job, Details:, J...|     9|
|[Our, Restaurant,...|    10|
|[Our, General, Ma...|    11|
|[Earning, potenti...|    12|
|[Dollar, General,...|    13|
|[Restaurant, Desc...|    14|
|[Who, We, Are, We...|    15|
|[A, Place, Where,...|    16|
|[Description, The...|    17|
|["Overview, Descr...|    18|
|[, seat, them, at...|    19|
+--------------------+------+
only showing top 20 rows
```

Figure 1: Example of Spark Dataframe

For computing efficiency it was decided to take a sample of the dataset, otherwise the Spark computing process takes a non-indifferent amount of time. The text of each description was preprocessed using the nltk package to remove punctuation, stopwords, extra space and to tokenize and transform all the text into lowercase. Although, there exist also the approach that uses stopwords in the creation of the shingles, in this case it was decided to not include them. Consequently, the RDD was created. An RDD is a Resilient Distributed Dataset and it is a data structure of Apache Spark where objects are represented as a collection that can be processed in parallel among clusters. It is immutable, it cannot be changed, meaning that its transformations create a new RDD; partitioned and fault-tolerant, meaning that Spark can recover RDD from node failures. In this project, the RDD was created selecting the previous dataframe using the `.rdd` attribute, in which each row object corresponds to a record in the dataframe. Then a transformation was executed: `.map`, which applies the lambda function to each row of the RDD. This transformation outputs a tuple of `doc_id` and the text of the job description referred to it.

2 Algorithm implementation

In the implementation of the algorithm, a Minhash signature was computed for each job description to find candidate pairs of similar job description using the measure of Jaccard similarity.

2.1 Shingling

A shingle, in the case of this dataset, is the substring of length k extracted from the words in the job description. The length of each shingle was chosen considering the one corresponding to the job descriptions and, finally, set to 3. The Shingling was performed on the rdd using `.flatMap`, which applies the lambda function to each element in the RDD and flattens the result. The function was applied for each document; it generates a list of tuples in which we have the document and the shingle extracted from the list of the sliced words, ensuring that it is done through all the document. Then, the result will give a flatten list of tuples with the document id and the shingle.
Then, the shingles were hashed to ensure that they fit within a 32-bit integer. In this

way, another RDD was created having as output a tuple consisting of the document id and the hash value of the shingle to fasten computations. All hashed shingles were concatenated with each unique document and then collected as a single list.

2.2 Minhashing

In computing the Minhash signature, a number of random hash functions were used to simulate random permutations. These hash functions were produced with some random parameters a, b and c and stored in a list. For each shingle, each of the defined hash functions computes a hash value. In summary, the minhash signatures were computed for each document, creating a smaller representation of the sets of the hashed shingles. The smaller values were retained for each hash function.

2.3 Locality-Sensitive Hashing

Locality-sensitive hashing (LSH) is an algorithm used to efficiently find similar pairs without having to look to every pair, but only considering the ones that are likely to be similar. This is done by hashing similar items into the same buckets. LSH was performed using the *MinHashLSH* class of Apache Pyspark, where the minhashes values were converted to vectors and the model was fitted into a dataframe. The banding technique was implemented. The number of bands is important because it establishes the partition of the minhash signature and it was set to 10. Each band contains a row number of the signature and if the two sets are similar, then at least one of their bands will hash to the same bucket. The threshold was defined as a function of the bands and rows and was estimated to be around **0.82**. Then, the MinHashLSHModel of Apache Pyspark was used in order to find similar job descriptions. The distance was computed according the *approxSimilarityJoin* method which states: Join two datasets to approximately find all pairs of rows whose distance are smaller than the specified threshold. Moreover, duplicate pairs and pairs that do not meet the similarity threshold were filtered out. The goal is to find pairs of documents that have similar MinHash signatures and by joining the dataframe that was created by the application of the MinHashLSH we are comparing each document's MinHash signature with every other document's signature.

3 Results

Finally, a Pandas dataframe was constructed, composed by three columns with two representing the documents and the third one as their jaccard similarity score showing that the task was achieved. Below a visual example:

	doc_id1	doc_id2	jaccard_sim
0	13395	1129756	0.122642
1	13395	1145791	0.151786
2	34993	584862	0.144068
3	273921	278072	0.205357
4	459967	1532570	0.085714
...	...	...	...
2770	278072	1621468	0.157895
2771	684947	1460189	0.205128
2772	922304	1038199	0.191667
2773	1532570	1621468	0.121495
2774	188603	872160	0.134454

2775 rows × 3 columns

Figure 2: Jaccard Dataframe

References

[1] A. Rajaraman e J. Ullman, Mining of Massive Datasets (2017)