Finding similar items

Algorithms for massive data
Master in Data Science for Economics

Rodion Horbunov
Matriculation number 988092
rodion.horbunov@studenti.unimi.it

1 Introduction

The goal of this project is to implement a detector of pairs (or sets of higher cardinality) of similar job descriptions. The detector analyzes the «LinkedIn Jobs & Skills» dataset, published on Kaggle and released under the ODC-By license, Version 1.0. The detector considers the job_summary column of the job_summary.csv file and outputs the pairs/sets of summaries inferred as similar. In the second paragraph, I will describe the data and dataset pre-processing techniques. During each paragraph, I would also highlight the issues I faced if any existed, and how I overcame them. In the third paragraph, I will describe the theoretic methodology behind used algorithm with practical usage from the dataset of the project. Consequently, I would describe developing and analyzing different solutions with respect to scaling it up. Finally, I would provide the experimental results obtained and point out the conclusions.

Before the project, I would like to draw some expectations and hypotheses. First of all, from the part of analyzing the dataset, I expect to find duplicates and other data issues, which is why I would like to put extra attention on the "uploading", and "investigating duplicates" steps. Due to the fact that the main type of data (job description) is the text, I expect that it is essential to apply at least basic text processing techniques. For example, I expect to deal with capitalization of text, with non-alphanumeric characters, and with the form of the words. During this stage, I would like to consider the regular expression, tokenization, lemmatization, and stemming. Also, I am planning to use the NLTK library for word tokenization, for imitating shingles built from words. During this step, I will try shingles size of 5. During the following steps, I will use minhashing to compress job descriptions into small signatures and preserve the expected similarity of any pair of documents.

Due to the fact that, I would like to leverage all the processes in default Colab for the comparison, I am expecting to have and analyze the computational difficulties. As a result, I would like to find an approach to calculate code on the full dataset or estimate the full dataset by processing a representative sample. At the conclusion of the project, I will evaluate and compare time spent during different sets of calculations and approaches. During this project, I would use Google Colab, with a System RAM of 51 GB, and a Disk size = of 225.8 GB.

2 Dataset and Data Preprocessing

The dataset has 1 297 332 rows, it has no duplicates in subset {job_link , job_summary}. However, if we are considering only the subset of {job_summary} it has only 958 192 unique values, which means that there are 339 140 identical job summaries (26.14%). These identical job summaries likely represent the same positions posted at different times due to such factors as company expansion, employee replacement, or renewing job posts for new applications. There are several possible ways are working with this dataset after this fact:

Approach 1: Group and Remove duplicates

1. Group the dataset by “job summary” column, and put in the list all “job link” that share the same summaries, because job summaries are absolutely similar or identical. As an alternative, to putting all “job link” in the list, we can put their indexes of the original data frame to reduce place taken.
2. After grouping, drop duplicate rows with identical job_summary values to reduce the dataset size. In the final table, we leave only the “job summary” with lowest index in the data frame.

Approach 2: Leave Dataset As-Is

1. If the dataset is too large for grouping, we can retain all rows without removing duplicates.
2. During the final stage of similarity calculations, identical job summaries will result in pairs with a maximum similarity score or minimal distance, equal to 0. To address this:

I would prefer Approach 1, due to the fact that even on the full-scale dataset of job descriptions removing duplicates is easily computed (up to 40 seconds). This approach reduced the working data set in a quarter, and make it more easier to process the data during the following steps. Due to the other techniques, my initial implementation plan is to start with 10 thousand rows of the original dataset for testing, and validation techniques, and to manage memory regarding the computational requirements. Once validated, I will scale the solution to the entire original dataset. Also during the “sample step” I can evaluate the time required during the main stage.

3 Algorithms, their implementations and description of the experiments

For this task, I would perform locality-sensitive hashing for documents. Using this approach, I could find the set of candidate pairs for similar documents and then discover the truly similar documents among them. Also, I should mention that this approach can theoretically produce false negatives – pairs of similar documents that are not identified as such because they never become a candidate pair. And, on the other hand, there will also be false positives – candidate pairs that are evaluated, but are found not to be sufficiently similar.

The most effective way to represent documents as sets, for the purpose of identifying lexically similar documents is to construct shingles from the document the set of short strings that appear within it. If we do so, then documents that share pieces as short as sentences or even phrases will have many common elements in their sets, even if those sentences appear in different orders in the two documents. [1, p. 78]

Define a k-shingle for a document to be any substring of length k found within the document. Then, we may associate with each document the set of k-shingles that appear one or more times within that document. The size of k should depend on how long typical documents are and how large the set of typical characters is. The important thing to remember is: that k should be picked large enough that the probability of any given shingle appearing in any given document is low. [1, p. 78]

A good rule of thumb is to imagine that there are only 20 characters and estimate the number of k-shingles as 20 in a power of k. For large documents, such as research articles, choice k = 9 is considered safe, for emails - 5 is okay. [1, p. 79]

For this case - I would provide an alternative form of character shingle that has proved effective, used for the problem of identifying similar news articles. The exploitable distinction for this problem is that the news articles are written in a rather different style than are other elements that typically appear on the page with the article. News articles, and most prose, have a lot of stop words, the most common words such as "and," "you," "to," and so on. In many applications, stop words are usually ignored, since they don't tell us anything useful about the article, such as its topic. During this approach in the news article more stop words would contribute in more shingles to the set representing the job description. Recall that the goal of the exercise is to find a similar job description, I assume that context as the stop words is important, therefore it can have a positive impact on the calculations. [1, p. 80]

As a result, to capture the context and to make computation easier, I would like to choose k large enough that the probability of any given shingle appearing in any given document is low, I picked a sequence of words with highly structured text as job description, with k equal to 5 (five words per shingle). I believe that this shingle size is optimal for this task, and that is why i will use it during the first iteration of the project. As it is mentioned before, I will not get rid of stop words and lemmatized the words, because I would like to save the context in the job description. Also, I replaced all non-alphanumeric characters, to provide better tokenization before following steps. I would like to mention, that there are possible points for growth in this part, I consider that it is possible to make shingles from characters, or to get rid of stop words, and to use lemmatization of words, for further comparing the empirical results and find better solution in case of accuracy and performance. Additionally, it is possible to experiment with a number of k value of shingles. I hashed shingles instead of using substrings directly. This step was performed to optimize taken place and increase the computation speed.

After providing the shingling, I choose the n number, which refers to the length of the MinHash signature for a document. To minhash a set represented by a column of the characteristic matrix, pick a permutation of the rows. The minhash value of any column is the number of the first row, in the permuted order, in which the column has a 1. Each document's MinHash signature is a vector of n values, where each value is the result of applying one hash function to the document's set of elements. The choice of n controls the trade-off between accuracy and computational cost. Larger n gives more accurate similarity estimates but requires more computation and storage. For my project I tried to use n equal to 128, but in the final steps I consider n equal to 32, due to the fact that this number of n increases the computations in four times.

The next step was to choose a threshold t that defines how similar documents have to be in order for them to be regarded as a desired "similar pair." Pick a number of bands b and a number of rows r such that br = n, and the threshold t is approximately (1/b) in a power of (1/r). If avoidance of false negatives is important, I could you may wish to select b and r to produce a threshold lower than t. For this exercise, I did not know real true positive and true negative pairs, which is why in the first stage I chose a threshold equal to 0.5, but after the first result, I increased it to 0.6 to avoid false negative pairs.

After that, I constructed the candidate pairs using the LSH technique, as a measure I used the Jaccard Similarity of sets. The Jaccard similarity of sets S and T is $|S \cap T|/|S \cup T|$, that is, the ratio of the size of the intersection of S and T to the size of their union.

As I planned, I also considered using the samples and a "straightforward" approach. I compared it with the parallel computing technique to leverage all my resources in the most effective way. Parallel Computing helped me to divide computational tasks into smaller, independent parts, that can be executed simultaneously across multiple processors or cores, speeding up processing time. Using the multiprocessing module in Python, I could leverage almost all the resources Colab provided. Based on the 10k sample, the straightforward approach took for about 60-80 sec, while multiprocessing - 24 seconds. Comparing the results on 100k sample, it took 15 minutes (900 seconds for the straightforward approach) and only 3 minutes (180 seconds) for the multiprocessing approach. That is why I decided to process the full-scale dataset in line with the multiprocessing approach, it was time-consuming but, all 1 mln rows were calculated for 40 minutes, which is not the worst case.

The very last step providing the similar pairs with the corresponding job description.

The next step was to perform all code in Apache Spark to compare the time per operation. The first option was - the word tokenizer with a sample size of 10 000 rows, and the second option is the sequence tokenizer with k equal to 3 due to the sample size of 100 000 rows. Word tokenizer performed all computations for about 25 seconds, that is the same good result as for the multiprocessing approach at a similar sample size. The second sequence tokenizer computations for 100 000 rows took for about 5 minutes, which is at least 3 times faster than previous approaches.

4 Conclusion

The goal of this project was successfully achieved by implementing a detector to identify pairs (or sets) of similar job descriptions from a large dataset. This involved the application of Locality-Sensitive Hashing (LSH) and computational optimizations for handling large-scale data. The key takeaways and areas of improvement are summarized below:

Implementation Success:

- A functional pipeline was built to preprocess job descriptions, generate k-shingles, compute MinHash signatures, and apply the LSH algorithm for approximate similarity detection.
- Using the Jaccard Similarity measure, candidate pairs were identified effectively, meeting the project requirements.

Efficiency and Scalability:

- Parallel computing significantly improved processing efficiency. During this approach, I could calculate pairs for a dataset of over 1 million rows. Also, it is more than in a five time more efficient a straightforward approach.

Optimization Decisions, that it is possible to improve and to experiment with:

- Shingle type
- Shingle size
- MinHash signature length
- LSH threshold

The choice to preprocess and deduplicate job summaries reduced the dataset size by 26.14%, streamlining further computations. Initial experiments with smaller dataset samples provided critical insights, guiding decisions for scaling up the solution. The Apache Spark optimized the algorithm more than 3 Times. The use of it should be considered as the breaking point for decreasing the computation time, although there can be possible difficulties with flexibility of the environment and data pre-processing.

Bibliography

1. Anand Rajaraman, Jeffrey D. Ullman, "Mining of Massive Datasets", second edition, 2010, 2011.