Algorithms for Massive Data
Project Report: Document Similarity

Hamed Davoodi, 11267A

13 Sep, 2024

Abstract

In this project, I focused on, using MinHash and locality-sensitive hashing (LSH), to find similar documents based on the kaggle's LinkedIn job summaries dataset. The idea was to simplify process that is prone to taking a lot of time especially because of the fact that the data is a big-data set. I used PySpark based on the fact that it is highly scalable for big data. This approach was advantageous since I was able to process all of those in parallel across the executors, which in the long run would utilize all the resources when the computation grows and also ensure scalability. First, pre-processing was done on the dataset to eliminate all job summaries that are not in English language and format the text in a way that there is no unnecessary punctuation. This was effective in eliminating unnecessary components on the data and simplified it to make it more organized. Also, instead of relying on the built-in PySpark functions, the logic is implemented here by using RDDs to have full control at every step. For the similarity check, I transformed each job summary into 2-word shingles which was nothing but splitting the sentences in every two words. After that these shingles were used to create a one-hot-encoded vectors and then I reduced these vectors with MinHash algorithm, not only keeping the inhereting features but also reducing the number of elements in each vector. I subdivided these hash vectors into 25 'bands', and applied LSH to bucketize similar documents. Each band was hashed separately in order to accumulate documents in sets of similar bands and group them together.

Contents

1 Introduction

Locality-Sensitive Hashing (LSH) is known for helping to find similar items in large datasets with a lot of dimensions. When it is needed to do tasks like comparing documents, old methods like comparing every pair of documents, can become hard when the dataset gets huge. LSH helps out with this by reducing the dimension of the data and avoid complex joining to compute combinations, but still keeping the similarities between documents mostly the same. It does this by breaking a signature matrix into smaller bands, so you can find similar pairs of documents quicker without having to check all the pairs only by implementing hash functions to map similar documents into same buckets.

MinHash, which is a big part of LSH, shrinks big sets of shingles (small text chunks) into smaller forms while still keeping the important similarity info from the original documents. MinHash makes several hash signatures for each document, and LSH takes this further by putting these signatures into bands, and then hashing them again. If two docs share a hash value in the same band, they're probably similar. This process cuts down the work needed, making LSH perfect for really large datasets. With big data growing, old ways of processing info can't keep up with the amount of data that's being created every day. Systems like Apache Spark have become super important for managing big data. PySpark, which is the Python version of Spark, helps manage big data by breaking the tasks across many machines, making sure even huge datasets can be dealt with without taking forever. Unlike the older ways, which often have issues scaling up and handling errors, PySpark is great when we need speed and reliability. Document similarity tasks often fall into the big data category since organizations might need to go over thousands or even millions of text docs. Comparing every doc to every other one is expensive and takes a lot of time. In these cases, LSH and MinHash help reduce the number of comparisons needed.

More precisely, PySpark's Resilient Distributed Datasets (RDDs) are what make it scalable. RDDs let users split data across many executors', and they're fault-tolerant, which means fast parallel processing. RDDs also give us control over how the data gets split up and processed, making them ideal for projects where optimization is key. For this project, I went with an RDD-based LSH setup to get more control over everything from the pre-processing of job summaries to the final document comparisons. By mixing LSH and MinHash with PySpark, I get a strong set of tools to handle document similarity in a big data setting. This makes the combination of LSH, MinHash, and PySpark really useful for things like content finding similar docs in big document sets. The next parts of this report will go through how I set up LSH on LinkedIn job summaries, using PySpark to handle the big dataset. The goal was to see how scalable this method is, while keeping accuracy in finding similar docs, especially in terms of precision. This project shows how distributed systems and efficient algorithms can deal with the problems that come with big data.

2 Methodology

In this section, I go over the steps involved in designing out the LSH algorithm to compare similar job summaries. In other words, the goal was to cut down the dimensions in which documents were compared and quickly spot similar pairs. First, I pre-processed the job summaries and made 2-word shingles (overlapping) for all the documents. Shingles are a common way of capturing a piece of a document by breaking it down into small fixed-size word groups. In this case, 2-word shingles were chosen to get a better way of tokenizing the job summaries (while it more general form shingling is applied on characters instead of words). To get

the signature matrix, I used MinHash with 100 different hash functions. These functions were applied to each document's shingles that encoded into zeros and ones (one-hot-encoding), giving me a brief summary that still captures how similar the documents are. There's what's called the 'signature matrix,' which is a pretty space-efficient representation of its related matrices. For this way of building the signature matrix, I split it into 25 bands. Each band had 4 rows of the matrix (100 rows in total, split into 25 bands = 4 rows per band). Each band gets hashed separately. If two documents have the same hash in any of the bands, they're likely to be matches. The banding technique narrows down the number of comparisons even more, speeding up the process by only checking documents that are probably similar. The documents that shared a hash value in any band were picked out, and those with the same hash were grouped as candidate pairs after hashing the bands. These pairs were then checked again to see if they were really similar, and I calculated their Jaccard similarity. The efficiency of LSH comes from how likely it is that two documents get mapped into the same buckets. This probability can be derived from the banding process and is given by the following formula:

$$P(\text{similarity}) = 1 - (1 - J^r)^b$$

Where:

- J is the Jaccard similarity between two documents.
- r is the number of rows per band (in this case, 4).
- b is the number of bands (25 bands).

This formula shows that as the Jaccard similarity between two documents increases, the probability of them being hashed into the same bucket also increases.

- **When J is high:** Two documents are very similar, and the probability of them ending up in the same bucket is high.
- **When J is low:** Two documents are not similar, and the probability of them being placed in the same bucket drops significantly.

This balance ensures that highly similar documents are likely to be found, while dissimilar documents are ignored, reducing the overall number of comparisons needed.

3 Results

To evaluate the effectiveness of the LSH implementation, as I mentioned before, I began by cleaning and preprocessing the job summaries. The initial step involved tokenizing each document into 2-word shingles, filtering out non-English texts, and refining punctuation. This resulted in a cleaned dataset ready for further processing. Once the data was cleaned, I applied MinHash using 100 hash functions to generate signatures for each document. Each MinHash signature represented a compressed form of the document's shingles while preserving the essential similarity characteristics between documents. The `minhash_rdd` contains the resulting MinHash signatures, which serve as the input for the LSH process.

[[(0, [14, 2, 6, 14, 6, 3, 1, 0, 21, 8, 2, 5, 19, 3, 1, 0, 9, 3,...]

After generating the signatures, I implemented LSH by splitting the signature matrix into 25 bands, each containing 4 rows. Each band was hashed independently, and documents that shared the same hash value in any band were grouped together into buckets. The final output was a set of bucket IDs, with each bucket containing document IDs that were highly likely to be similar. The results from this step are crucial for reducing the number of pairwise comparisons needed to identify similar documents.

Bucket 291200: Documents [1, 4, 5, 9, 10, 11, 12, 15, 16, 17, 19, 21, 22, 23, 26, 27, 28, 33, 39, 41, 42, 46, 47, 48, 49]
Bucket 724851: Documents [2, 40]
Bucket 959101: Documents [15, 17, 18, 24, 25, 35, 40, 45]
Bucket 208701: Documents [24, 34, 44]
Bucket 473552: Documents [1, 7, 16, 17, 23, 37, 45]
Bucket 298152: Documents [3, 16, 17, 25, 32, 37, 38]
Bucket 679252: Documents [8, 29, 37]
Bucket 346852: Documents [8, 25]
Bucket 559052: Documents [14, 17]
Bucket 592252: Documents [15, 21]
Bucket 795702: Documents [15, 23, 27, 48]
Bucket 751652: Documents [29, 32]
Bucket 72202: Documents [32, 37]
Bucket 301753: Documents [26, 33]
Bucket 364053: Documents [30, 40]
Bucket 704804: Documents [0, 29]
Bucket 332004: Documents [2, 38]
Bucket 514905: Documents [10, 31]
Bucket 839956: Documents [2, 3, 20, 24, 31, 34, 36, 38, 40, 43, 44, 45]
Bucket 195707: Documents [5, 9, 19, 41, 46]

To further evaluate the accuracy of the LSH algorithm, I sampled pairs of documents from the same bucket and computed their Jaccard similarity. The Jaccard similarity compares the overlap between the two documents' shingles, providing a numeric value between 0 (no similarity) and 1 (complete similarity). then, I compared this computed similarity to the actual text of the documents to assess whether the algorithm successfully grouped similar documents.

For example, comparing two documents with IDs `0` and `28`, I obtained the following Jaccard similarity:
Approximate Jaccard Similarity from MinHash between doc1 = 0 and doc2 = 28: 0.5847750865051903
Below are the actual texts of the two documents:

- **Document 0:** `rock n roll sushi is hiring a restaurant manager! as our restaurant manager, you'll never be bored. you'll be responsible for making sure our restaurant runs smoothly. we offer competitive compensation insurance benefits bonus opportunities a great work atmosphere duties/responsibilities ensuring that our restaurant is fully and appropriately staffed at all times maintaining operational excellence so our restaurant is running efficiently and effectively ensuring that all laws, regulations, and guidelines are being followed`

creating a restaurant atmosphere that both patrons and employees enjoy various other tasks as needed requirements previous experience as a restaurant manager extensive food and beverage knowledge, and the ability to remember and recall ingredients and dishes to inform customers and wait staff great leadership skills familiarity with restaurant management software demonstrated ability to coordinate a staff show more show less

- **Document 28:** the ideal candidate has a passion for food and beverage, a genuine ability to connect with guests and staff alike, and brings the core value of 'team' to the restaurant. a natural leader, you are responsible for maintaining the highest levels of hospitality while leading service on the floor. responsibilities supervise day-to-day activities and assist in the food and beverage outlets create innovative programs and promotions that drive revenue through increased guest patronage aid in all financial budgeting operations to maximize profitability qualifications at least 1-2 years' of full-service restaurant management flexibility in working hours and a willingness to cover shifts as needed ability to multi-task, organize, and prioritize work show more show less

By going through the Jaccard similarity scores alongside the actual text content, I was able to get a pretty good idea of how well the LSH approach grouped documents that were truly similar in meaning. For those document pairs that had high Jaccard scores, the content was indeed very closely related, almost as if they were talking about the same thing. On the other hand, the pairs with lower similarity scores didn't have much in common at all. It became clear that LSH was doing a solid job of picking out the documents that shared strong similarities, even though it wasn't perfect in every case. I decided to godeeper into how well LSH was performing by using precision as a way to measure it. What I did was compare the pairs of documents that LSH flagged as similar with their actual similarity based on their Jaccard scores. Precision, in this sense, was about figuring out how many of those flagged pairs were correctly grouped as similar, relative to the total number of pairs that LSH picked out. It's a useful way to see if the algorithm was doing a good job without catching too many false positives—those are the document pairs that got grouped together even though they didn't really have much in common. While it wasn't perfect, it showed that LSH could reliably find the true similarities, and any errors it made were relatively few and far between.

4 Conclusion

The effectiveness of the LSH approach was worked by balancing the trade-off between false positives and false negatives, which relied on factors like the number of hash functions and the number of bands. Even though using more hash functions could lead to more time and resources being spent on the hash tables, they help cut down on false negatives, improving the precision in finding true document pairs. On the hand, when I start adding more hash functions, it can increase up computing time and memory space, which isn't great. Similarly, when I reduce the number of bands, it helps me to find more similar documents. But at the same time, the chances of false positives,where dissimilar documents get grouped together, increases.

One big plus here is the scalability and flexibility of the LSH method, especially when you're working in big data environments like PySpark. Using RDDs gave me a lot more control over how things got processed and freed me from issues like data skew that can come up in the MapReduce model. This also pointed out the importance of partitioning and pre-processing the data to boost scalability. The results of the work showed

that LSH really does offer an efficient way to find similarities in documents on a large scale, particularly in big data analyses. Plus, it's possible to tweak the method even further for better performance, especially when setting the parameters based on the type of dataset we're working with.

5 Reference

Leskovec, J., Rajaraman, A., & Ullman, J. D. (2014). *Mining of massive datasets.* Cambridge University Press.