UNIVERSITÀ DEGLI STUDI DI MILANO

Finding Similar Items in Job Descriptions

Algorithms for Massive Data Course

Shojaat Joodi Bigdilo

June 2024

Abstract

In this project, we tackled the challenge of simplifying the job search process by implementing advanced data processing techniques on a large-scale dataset. Using the LinkedIn Jobs & Skills dataset, we focused on identifying pairs of similar job descriptions through the application of shingling, MinHashing, and Locality Sensitive Hashing (LSH), all facilitated by PySpark's robust data handling capabilities. Our approach involved an extensive exploratory analysis, including text preprocessing methods to improve the performance of processing. By employing LSH for approximate similarity joins with Jaccard distance we efficiently detected similar job descriptions across small chunk of dataset to avoid high computation time which needs for whole corpus.

1. Introduction

In the modern job market, the vast number of opportunities on online job search platforms can be overwhelming for both job seekers and recruiters. While these platforms offer extensive access to job listings, candidates must navigate through numerous descriptions to find suitable roles, and recruiters must sift through countless applications to identify the best candidates.

Our project aims to simplify this process by detecting similar job descriptions using advanced data processing techniques. We focus on the job summary column of the job_summary.csv file from the LinkedIn Jobs & Skills dataset, available on Kaggle. The goal is to identify pairs or sets of similar job descriptions, providing valuable insights for both recruiters and job seekers.

We begin by preprocessing the text data from job postings using techniques such as shingling, MinHash signature generation, and Locality Sensitive Hashing (LSH) with Jaccard similarity. These methods convert the data into a format suitable for comparison, facilitating the analysis of similarities between job summaries.

Apache Spark, an efficient framework for large-scale data analysis, underpins our project. Utilizing Spark's Sparce Vector, Spark SQL for structured data processing, and Spark **MLlib** library for feature extraction and similarity analysis, we efficiently handle and process the large dataset.

2. Dataset Description

The dataset utilized in this project is the "LinkedIn Jobs & Skills" dataset, obtained from Kaggle. It contains a variety of job summaries and related information, with the job_summary column being the primary focus for analysis. Each row contains a job description/summary.

3. Theoretical Framework

3. 1. Shingling:

Shingling involves breaking down the text into overlapping sequences of words or characters, known as shingles.

K-shingle:

A K-shingle (or K-gram) is a sequence of k tokens extracted from a document, where the token can be a character, word, or other text unit, depending on the application. It's important to select a k value large enough to ensure that the probability of any given shingle appearing in a document is low. Generally, K=5 is used for short documents and K=10 for long documents.

For example, if k=2 and the text is "job description posted on LinkedIn," the word-based 2-shingles would be ["job description", "description posted", "posted on", "on LinkedIn"]. K-shingles are primarily used to assess document similarity. By comparing the sets of shingles from two documents, one can estimate their similarity.

3.2. **Jaccard similarity**

Jaccard similarity measures how similar two sets are by dividing their intersection's size by the size of their union. Given sets S1 and S2, the Jaccard similarity ranges from 0 to 1, where 0 indicates no shared elements and 1 indicates identical sets.

3. 3. MinHashing

Minhashing is the next step in our process, which allows us to convert sparse vectors into compact signature vectors preserving the information about their similarity. Starting with our sparse vectors, we randomly generate one minhash function for each position in the signature vector (i.e., the dense vector). For example, to create a signature vector of 50 numbers, we would use 50 minhash functions.

These MinHash functions involve a randomized order of numbers, counting from 1 to the length of the vocabulary. Because the numbers are randomized, a specific number (e.g., 1) might appear in a different position, such as position 67, within a given MinHash function.

This process is repeated for each position in the signature vector, producing multiple values for the MinHash signature by assigning a unique minhash function to each position.

3.4. Locality Sensitive Hashing (LSH)

Locality Sensitive Hashing (LSH) involves hashing items multiple times so that similar items are more likely to be placed in the same bucket compared to dissimilar items. By focusing on pairs that hash to the same bucket, we reduce the number of comparisons needed for similarity checks, ideally minimizing false positives (dissimilar pairs hashed together) and false negatives (similar pairs not hashed together). A common method uses minhash signatures, where the signature matrix is divided into bands of rows, each hashed into multiple buckets. This approach ensures that similar items are likely to hash together in at least one band while maintaining efficiency.

4. Data Preprocessing

4.1. Data Organization

The dataset is imported and organized within a PySpark DataFrame to enable efficient handling and processing of large data volumes. because it is designed to handle large datasets that cannot be fitted by a single machine's memory, enabling data processing across multiple nodes in a distributed manner.

4.2. Preprocessing Techniques

Text data from varied sources like LinkedIn can be disorganized, unstructured, and highly diverse, often plagued by inconsistent formatting, spelling errors, and special characters. Preprocessing is necessary to standardize this text data into an optimal format for analysis, especially for similarity detection.

As a first step, we read CSV file which was a PySpark dataframe. The dataframe consists of 1,264,216 records and 2 columns (job_link and job_summary). For further analysis, we dropped the columns job_link, and then the ID column was added to DataFrame.

4.2.1. Missing and unique values

As a next step, we checked if there are any missing values in the columns and counted the numbers of unique values. There are no missing values in the dataframe.

4.2.2. Duplicate Documents

By counting distinct text, the result shows that we have 958192 Distinct texts, and 339140 document are Duplicate. Therefore, these duplicates are eliminated from dataset to avoid extra computation.

Table 2: Counting number of distinct texts

Column names	Count unique values	Number of duplicate Document
Id	1297332	0
job_summary	958192	339140

For example, checking the result of duplicate columns shows that document which contains 'job_summary' starts with "Job Title:\nCertified Nursing Assistant (CNA)\nCompany: . . ." has repeated 66 times in the dataset.

4.2.3. Text Cleaning

For this step Pyspark UDF function is used to apply the function to every entry in the column. By taking this approach we can effectively transform text data within a Spark DataFrame preparing it for analysis or machine learning tasks.

- Transforming to lowercase
- Removing HTML Tags
- Removing URLs
- Removing extra space
- Punctuation Removal:
- Removing numbers

Removing numbers from tokens to retain only words. For example, a document with id = 160 has 42 different numbers inside itself, so these numbers must be deleted since they do not have meaning in computation and they just increase the number of shingles.

- **Removing Non-ASCII characters:**

Some texts have some non-ASCII characters like (ã°â\x9fâ\x9fâ¡), so we need to delete them from texts. For example, document with id = 915 has 5 non-ASCII characters inside itself, so these characters must be deleted since they do not have meaning in computation and they just increase the number of shingles. Characters found inside document with id = 915 are: ð\x9f\x9f¡, ð\x9f\x9f¡, ð\x9f\x9f¡, ð\x9f\x9f¡ , ð\x9f\x9f¡ , â£

- **Tokenization**

Then we split sentences into sequences of words using “Tokenizer” method from the PySpark machine learning module (MLlib (DataFrame-based)).

- **Removing StopWords**

As the next step, we eliminated stop words from the token list using the “StopWordsRemover” function from PySpark MLlib module. We employed the default stop word list provided by Spark, which includes words such as 'i', 'me', 'my', 'myself', 'we', 'our', 'ours', and others.

Additionally, we counted the number of tokens in each document both with and without the removal of stop words, as shown in the following table. It is evident that in some cases, stop words make up 10-35% of the tokens in the Job descriptions. The token lists for both scenarios are also displayed in the table.

Table 2: Counting the number of tokens after Stopwords removed

Id	Number of tokens	Number of tokens After Stopwords removed	Number of removed Stopwords
160	1917	1366	-551
349	800	497	-303
589	269	151	-118
1319	362	251	-111
1324	303	219	-84
1803	338	203	-135
1928	1337	889	-448

5. Algorithm Implementation and Results

5.1. Shingling:

A custom function ‘shingles()’ was implemented to generate shingles from job summaries. This shingles function generates shingles of a specified length (k=2 in this case) and stores them as sets to ensure uniqueness.

I chose K=2 for two main reasons. **First**, if we use single-word shingles, given that the common English vocabulary consists of around 3000 words, it's highly likely that two documents will share at least one word, making them candidate pairs easily. Using two-word shingles helps avoid this issue. **Second**, certain vocabularies are meaningful only when they occur together. For instance, "United States" has a different meaning when split into "United" and "States". Other examples include "remote work," "office work," "health insurance," "sales manager," "highly professional," "Python programming," and "software engineer." By using two-word shingles, we capture these meaningful phrases accurately.

Finally, we flatten the shingles column to get all unique shingles to create an index for each shingle.

5.2. Sparce vector and Efficiency:

In a sparse vector representation, only the non-zero elements are stored. This is to save space and computational resources. A sparse vector is typically represented in the format (size, [indices], [values]), where:

- size is the total number of dimensions.
- indices are the positions of the non-zero elements.
- values are the corresponding non-zero values.

The rest of the dimensions, which are implicitly zero, are not stored.

For example, for document with Id = 4427 sparce vector is Vectors.sparse((558004, [36, 3459, 3460, …], [1.0, 1.0, 1.0, …])). This means there are 558004 shingles in whole corpus, but this document contains shingles with index numbers of 36, 3459, 3460 and so on. All non-zero values are treated as binary "1" values. Since this document contains 93 shingles, the length of [indices] and [values] are 93 rather than 558004.

Scalability and Efficiency of the Proposed Solution

The proposed solution leverages sparse vectors to represent high-dimensional data efficiently. This approach significantly improves the scalability of the solution, especially when dealing with large datasets. Here's how the solution scales up with data size:

Memory Efficiency

Since Sparse vectors are designed to store only the non-zero elements and their indices, rather than all elements. Therefore, the memory required to store a sparse vector is significantly less than that required for a dense vector.

- **Dense Vector:** Each document represented as a vector of all possible words or shingles (in our project 558004 shingles) requires storing 558004 float values per document.
- **Sparse Vector:** Each document only stores the indices and values of words present in it, significantly reducing the memory footprint.

Example Comparison for Document with Id = 4427:

- **Sparse Vector:** (558004, [36, 3459, 3460, ...], [1.0, 1.0, 1.0, ...])
- **Dense Vector with size** 558004**:** [0.0, 0.0, 0.0, ..., 1.0, ..., 0.0, 1.0, 0.0, ..., 0.0]

For Document with Id (**4427**) which is a vector with 558004 dimensions, it has only 93 shingles, so if only 93 elements are non-zero, the memory usage of the sparse vector is primarily the sum of the sizes of 93 indices and 93 values, plus a single integer for the size attribute. In contrast, the dense vector would need to store 58004 float values, most of which are zeros. memory size of Document with Id = 4427:

- **Memory size of the sparse vector**: 6.79 KB
- **Memory size of the dense vector**: 17,437.68 KB

This shows that the dense vector consumes approximately 2569 times more memory than the sparse vector. This drastic reduction in memory usage demonstrates the efficiency and scalability of using sparse vectors for high-dimensional, sparse data, making our solution feasible for large-scale applications.

Computational Efficiency

When processing large datasets, computational efficiency is crucial. Sparse vectors not only save memory but also improve computational efficiency by reducing the number of operations needed. Operations on sparse vectors can be restricted to non-zero elements, which is faster than processing all elements in a dense vector.

5.3. MinHashLSH function

MinHashLSH is a Locality Sensitive Hashing (LSH) scheme for Jaccard distance metric. This Function transforms the input feature vectors to multiple hash values. The input features are sparse vectors which it is calculated in the last step for each document (job summary).

The formula is `MinHashLSH`(***inputCol***, ***outputCol***, ***seed***, ***numHashTables: int = 1***). In this model the parameter named ***numHashTables*** determines the number of hash tables (hash functions). A higher number of hash tables can reduce the false negative rate missed similar items, but at the cost of increased computational overhead and memory usage.

In this case, 20 hash tables are used to strike a balance between accuracy and performance. Each hash function creates one hash value for each document. Since our number of hash functions are 20, we have 20 hash values per document. The MinHashLSH function creates these values for each document, and the result is Signature Matix or Hash Table which will be as an input data for "**approxSimilarityJoin**" function in order to apply similarity.

Result of Signature Matix or Hash Table for document id = 160:

```
[DenseVector([817945.0]), DenseVector([84975.0]), DenseVector([1314580.0]), DenseVector([792147.0]),
DenseVector([1059460.0]), DenseVector([3875662.0]), DenseVector([641826.0]), DenseVector([482020.0]),
DenseVector([1434890.0]), DenseVector([1621684.0]), DenseVector([2809550.0]), DenseVector([413624.0]),
DenseVector([938124.0]), DenseVector([1989136.0]), DenseVector([726703.0]), DenseVector([5337888.0]),
DenseVector([546638.0]), DenseVector([2574512.0]), DenseVector([85776.0]), DenseVector([2619402.0])]
```

ApproxSimilarityJoin Function:

The MinHashLSH model also supports approximate similarity join between two datasets by using the "approxSimilarityJoin" function. This computes the locality sensitive hashes for the input rows, then perform an approximate similarity join between the transformed DataFrame and itself, calculating the Jaccard distance between the vectors (job_summary). The results are filtered to exclude self-pairs and only include pairs with a Jaccard distance below a specified threshold (0.6 in this case). The input for this function is two datasets, in this case, both datasets are Signature Matix (Hash Table) computed in the last step by `MinHashLSH` method.

`approxSimilarityJoin`(***datasetA***, ***datasetB***, ***threshold: float***, ***distCol: str = 'distCol'***)

6. Result

The dataset was processed in small chunks to manage the large volume of data efficiently. The chunk was analyzed to identify similar job summaries. The system successfully identified pairs of similar job descriptions, demonstrating the effectiveness of the implemented methodology.

Key results from the analysis are as follows:

Table of Similar Job Summaries

The following table shows the top 6 rows of a dataset sorted by Jaccard Distance in ascending order. Each row includes three columns: idA, idB, and JaccardDistance.

idA: The identifier of the first document in the pair being compared.

idB: The identifier of the second document in the pair being compared.

JaccardDistance: The calculated Jaccard Distance between the two documents. This value ranges from 0 to 1, where a lower value indicates higher similarity, and a higher value indicates lower similarity.

Table 3: Similarity of pairs of documents with distance < 0.6 sorted in ascending order

idA	idB	JaccardDistance
503	948	0.0
107	406	0.0
544	963	0.0
1623	2337	0.0
1213	1812	0.0
1785	2221	0.0

This row indicates that the pair of documents with IDs 503 and 948 has a Jaccard Distance of 0, suggesting that these two documents are identical in terms of the elements being compared, as a Jaccard Distance of 0 indicates complete similarity.

Distance	ID	Job Summary
0	503	“Description\nOur\nRestaurant Team/Shift Leaders\nhave a dual role - youÃ¢Â\x80Â\x99ll serve as both a restaurant leader and a team member. As a leader, youÃ¢Â\x80Â\x99ll work closely with the Restaurant Manager ensuring all operating procedures are followed. YouÃ¢Â\x80Â\x99ll also assist with scheduling, training and supervising Team Members to ensure each customer enjoys a hot, freshly-prepared product using the highest quality ingredients served in a comfortable, clean, friendly environment.\nWhat's In It For You\nCompetitive Weekly Pay\n$15.25 - $17 / hour\nSchedule Flexibility Ã¢Â\x80Â\x93 Day/Evening/Overnight Shifts\nDiscounted Meals\nOpportunities for Career Development and Growth\nWhataburger Family Foundation and Scholarship Program\nMedical, Dental and Vision Plans\n401K Savings Plans\nWhatagames (Ask us about this!)\nOur people make the difference

		at Whataburger. We take pride in our work, take care of each other and love serving our customers. Each and every day youÃ¢Â\x80Â…”
	948	“Description\nOur\nRestaurant Team/Shift Leaders\nhave a dual role - youÃ¢Â\x80Â\x99ll serve as both a restaurant leader and a team member. As a leader, youÃ¢Â\x80Â\x99ll work closely with the Restaurant Manager ensuring all operating procedures are followed. YouÃ¢Â\x80Â\x99ll also assist with scheduling, training and supervising Team Members to ensure each customer enjoys a hot, freshly-prepared product using the highest quality ingredients served in a comfortable, clean, friendly environment.\nWhat's In It For You\nCompetitive Weekly Pay\n$16 - $17.50 / hour\nSchedule Flexibility Ã¢Â\x80Â\x93 Day/Evening/Overnight Shifts\nDiscounted Meals\nOpportunities for Career Development and Growth\nWhataburger Family Foundation and Scholarship Program\nMedical, Dental and Vision Plans\n401K Savings Plans\nWhatagames (Ask us about this!)\nOur people make the difference at Whataburger. We take pride in our work, take care of each other and love serving our customers. Each and every day youÃ¢Â\x80Â…”

Table 5: Similarity of pairs of documents with distance < 0.6 sorted in descending order

idA	idB	JaccardDistance
3284	4955	0.5998985801217038
4051	4261	0.5997536945812808
3187	4189	0.5995115995115995
3771	4880	0.599294947121034
3817	3882	0.599224305106658
3771	4808	0.599121361889072

The first row indicates that the pair of documents with IDs 3284 and 4955 has a Jaccard Distance of 0.59, suggesting a relatively low level of similarity between these two documents, as a higher Jaccard Distance value (closer to 1) indicates greater dissimilarity.

Table 6: Pairs of documents with Jaccard Distance between 0.2 and 0.3

idA	idB	JaccardDistance
1909	3014	0.2005494505494505
3058	3148	0.2008196721311475
2023	2036	0.20105820105820105
1602	2205	0.20110192837465568
2205	2223	0.20110192837465568
553	1909	0.20110192837465568

Comparison of Some Document Pairs Based on Jaccard Distance

In this part, one document as an example from each of above tables will be analysed by the number of words contained inside each of them to get more information about similarity of documents.

Table 7: Examples of the similar document pairs with number of their words

	Id = 1909	Id = 3014	Id = 503	Id = 948	Id = 3284	Id = 4955
Jaccard Distance	0.20		0		0.59	
Jaccard Similarity	0.8		1		0.41	
Number of words After Stopword removed	344	361	346	346	1876	1387
Number of Unique words in document	249	260	268	268	718	634
Number of common Unique words	240	240	268	268	481	481
Percentage of common words in Documents	96 %	92 %	100 %	100 %	66 %	75 %

Documents with IDs 503 & 948 (Jaccard Distance: 0)

- **Jaccard Similarity**: 1 (indicating complete similarity).
- **Number of Words After Stopword Removal**: Both documents have 346 words.
- **Number of Unique Words**: Both documents contain 268 unique words.
- **Number of Common Unique Words**: 268 (all unique words are common).
- **Percentage of Common Words**: 100%.

The documents with IDs 503 and 948 are identical in content, sharing all their unique words. This indicates they are either duplicates or have exactly the same topic and words.

Documents with IDs 3284 & 4955 (Jaccard Distance: 0.59)

- **Jaccard Similarity**: 0.41.
- **Number of Words After Stopword Removal**: Document 3284 has 1876 words, while document 4955 has 1387 words.
- **Number of Unique Words**: Document 3284 has 718 unique words, while document 4955 has 634 unique words.
- **Number of Common Unique Words**: 481.
- **Percentage of Common Words**: 66% (document 3284), 75% (document 4955).

The documents with IDs 3284 and 4955 have a moderate level of similarity, sharing a significant portion of their unique words but also having a substantial amount of unique content. This suggests that while they cover related topics, they differ considerably in their wording and possibly in their focus.

Documents with IDs 1909 & 3014 (Jaccard Distance: 0.20)

- **Jaccard Similarity**: 0.8.

- **Number of Words After Stopword Removal**: Document 1909 has 344 words, while document 3014 has 361 words.
- **Number of Unique Words**: Document 1909 has 249 unique words, while document 3014 has 260 unique words.
- **Number of Common Unique Words**: 240.
- **Percentage of Common Words**: 96% (document 1909), 92% (document 3014).

The documents with IDs 1909 and 3014 are quite similar, sharing most of their unique words. This high degree of overlap indicates that they discuss very similar topics with slightly different wording or additional unique content in one of the documents.

Comparing Dance and Sparse Vector result:

MinhashLSH operation with 5000 documents, if we use shingles_to_one_hot_vector, the Computation time is 17.329. while, if we use shingles_to_sparse_vector, Computation time is 7.484, around 2.5 times faster.

Discussion

When we increase the number of documents from 50000 to 10000, also 20000, 50000, and 100000, even if I don't talk about 1 million documents, in this situation, the MinhashLSH function is calculated in some seconds, Also the " approxSimilarityJoin" function will be calculated in some seconds. But, the problem is when you want to call result_filtered.show(10) function to show the result of JaccardDistance for 10 document pairs, unfortunately, this function does not show the result immediately, it takes some hours and finally gives an error. the problem is when it runs to show the result, the capacity of the disk in Google Collab increases enormously for example, from 33 GB to 76 GB. This increase causes an error and stops the whole process because the capacity of Google Collab's disk becomes full.

This shows that when we run the result_filtered.show(10) function, when we want to call the data, first, it stores all of the calculated data in the disk, then gives the result, that's the reason when we call the show() function to show the result of 10 pair document, it won't show the result, because I think first, because it brings data from the Spark distributed environment into the local driver memory. As a result, I couldn't find results for a dataset with more than 5000 documents.

Conclusion

This project employs a sophisticated strategy to handle large datasets effectively by processing them in manageable chunks. By combining Locality Sensitive Hashing (LSH) and MinHash signatures, the implemented code accurately identifies similar job descriptions with high efficiency. However, collecting and showing the result for a big corpus of more than 5000 documents is not possible, because of disk limitations in Google Collab and also some other errors that happen while calling the result to show them.

References

1. Mining of Massive Datasets, A. Rajaraman and J. Ullman.

2. PySpark Documentation.

3. Kaggle: LinkedIn Jobs & Skills Dataset.

4. https://medium.com/@imeshadilshani212/words-as-vectors-sparse-vectors-vs-dense-vectors-18e2084ad312

5.https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.linalg.SparseVector.html#pyspark.ml.linalg.SparseVector

6.https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.feature.MinHashLSH.htm

7.https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.feature.MinHashLSHModel.html#pyspark.ml.feature.MinHashLSHModel.approxSimilarityJoin

8.https://towardsdatascience.com/similarity-search-part-5-locality-sensitive-hashing-lsh-76ae4b388203