Finding similar items

Dzmitry Kurch, DSE

Algorithms for massive data project, June 2024

1. Introduction

The primary goal of this project is to detect similar pairs of records or identify a set of similar records of higher cardinality within a dataset. Specifically, the similarity detection is focused on the *job_summary* column containing LinkedIn position descriptions from the Kaggle dataset *Linkedin Jobs & Skills*. An essential application of such techniques for detecting similarities among documents is the deletion of duplicate records referring to the same entity when merging two or more data sources. This process is typically known as entity resolution or record linkage. Other terms often used in this context include entity disambiguation, entity linking, duplicate detection, deduplication, record matching, and etc. By identifying these similar records, we can ensure data integrity and avoid redundancy, which is crucial for maintaining clean and reliable datasets.

For this project, I utilized the Google Colab platform *https://colab.research.google.com/drive/10hRxCvqAu8IRQBlHp2DJJafmqNxHH0Zv?usp=sharing* and Spark with the Python API (PySpark). To handle the text preprocessing tasks, I employed the spark-nlp library, which provided robust tools for processing the job summaries. For the clustering algorithms necessary to group similar records, I used Spark's MLlib library.

One of the significant advantages of using Spark for this project is its inherent scalability. Spark's distributed computing capabilities ensure that the solution can handle large datasets efficiently. All operations, from text preprocessing to clustering, are performed in a distributed manner, making the solution capable of scaling with respect to the dataset size out-of-the-box. This makes this approach not only effective but also highly efficient for large-scale data integration and entity resolution tasks.

2. Data organization and preprocessing

The data for this project was originally stored in a CSV file containing 1.4 million records with two columns: job link and job summary. Given the large size of the dataset and the limitations of the public Google Colab cluster, I decided to work with a sample containing 10.000 rows of the original data. This was necessary because the community edition lacks the computing power and number of nodes required to efficiently process the entire dataset within the available time constraints.

To start, I loaded the data directly into Colab via Kaggle API as csv file and extracted the random sub-sample of it. After that I loaded into Spark Dataframe via *spark.read.csv()* for further preprocessing. Using Spark-SQL allowed me to efficiently query and manipulate the sample data before diving into more detailed text processing tasks.

For text preprocessing, I utilized the spark-nlp library, which offers a comprehensive set of tools for handling text data. The preprocessing pipeline included several key steps:

1. **Document Assembly**: Converting the raw text into a structured format suitable for further processing.
2. **Tokenization**: Splitting the text into individual words or tokens.
3. **Normalization**: Standardizing the text by converting it to lowercase and removing any special characters or punctuation.
4. **Stop Words Removal**: Eliminating common words that do not carry significant meaning (e.g., "and", "the").
5. **Stemming**: Reducing words to their root forms to ensure that different forms of the same word are treated equally (e.g., "running" becomes "run").
6. **Finishing**: Converting the processed tokens back into a format that can be used for further analysis.

These preprocessing techniques are essential for cleaning and standardizing the text data, which helps improve the accuracy and efficiency of subsequent analytical steps.

Example of top-10 most frequent tokens extracted: work, experi, manag, require, team, provid, include, service, show, care.

Following the text preprocessing, I applied two methods for transforming the text data into numerical features: Hashing Term Frequency (TF) and Inverse Document Frequency (IDF) using Spark's MLlib library. Hashing TF provides a bag-of-words (BOW) representation, where each document is represented by the frequencies of its tokens. It reduces the dimensionality of the feature space by mapping tokens to a fixed number of hash buckets, which helps mitigate the issue of high memory consumption and computational complexity associated with large vocabularies. IDF then scales these frequencies based on the importance of each token across the entire dataset, helping to highlight more informative words. As a result, I ended up with two possible ways of sentence embedding: TF-IDF and BOW. These embeddings served as the foundation for detecting similarities among the job summaries.

3. Finding similar pairs

To identify similar pairs of records, I began by performing a cross join of the table with itself. This “brute-force” operation compares every record with every other record in the dataset. To avoid redundant comparisons, I filtered out the duplicated pairs, ensuring that each pair (A, B) and (B, A) was represented only once. This preparation step set the stage for row-wise similarity calculations. Having a computing power constraint I used (100, *sample_size)* cartesian product, i.e. I was looking for similar records in all data sample across 100 random job postings.

For measuring similarity, I employed two custom User Defined Functions (UDFs): one for calculating Jaccard similarity and the other for computing cosine similarity.

Jaccard Similarity is a measure of similarity between two sets, defined as the size of the intersection divided by the size of the union of the sets. In the context of text data, this means comparing the sets of tokens (words) in each document. The formula for Jaccard similarity is:

$$\textit{Jaccard Similarity} = \frac{|A \cap B|}{|A \cup B|}$$

where *A* and *B* are the sets of tokens from two documents. A higher Jaccard similarity indicates a greater overlap between the two sets, thus higher similarity.

Jaccard similarity	Number of pairs detected
100%	64
80%-99%	39
60%-79%	12
40%-59%	172
20%-39%	28443
<20%	494460

Table 1 - Top pairs of documents by Jaccard similarity

In the table above you may see the result of similar pairs mining where in the left column I reported the ranges of Jaccard similarity between items and in the right column there are number of pairs detected. We can see that there are 64 pairs of records with identical set of tokens and in total there are approximately 200 pairs with more than 50% Jaccard similarity.

Cosine Similarity, on the other hand, measures the cosine of the angle between two non-zero vectors in a multidimensional space. When applied to text data, the vectors are typically the TF-IDF representations of the documents. The formula for cosine similarity is:

$$\textit{Cosine Similarity} = \frac{\vec{A}\vec{B}}{\|\vec{A}\|\|\vec{B}\|}$$

where A and B are the TF-IDF vectors of two documents and $||A||$ and $||B||$ are their magnitudes. Cosine similarity ranges from -1 to 1, with 1 indicating identical direction (maximum similarity), 0 indicating orthogonality (no similarity), and -1 indicating opposite direction (maximum dissimilarity).

Cosine similarity	Number of pairs detected
100%	64
80%-99%	53
60%-79%	142
40%-59%	359
20%-39%	6472
<20%	516100

Table 2 - Top pairs of documents by Cosine similarity

Following the same logic of the table for the Jaccard Similarity groups (Table 1) in the Table 2 above you can see how many pairs there within different ranges of cosine similarity are.

When it came to identifying 100% duplicates, both Jaccard similarity and cosine similarity yielded the same pairs of records. This consistency is since completely identical documents will have maximum similarity across both metrics. However, beyond perfect duplicates, the results varied because the metrics capture different aspects of similarity.

Jaccard Similarity focuses on the presence or absence of tokens, making it sensitive to the overlap of unique words between documents.

Cosine similarity considers the frequency and importance of words through TF-IDF, capturing the angle between document vectors in a high-dimensional space, which reflects the distribution of word importance.

These differences mean that Jaccard similarity might be more suitable for comparing documents with similar sets of words, while cosine similarity is often better at capturing the overall similarity in terms of word frequency and significance.

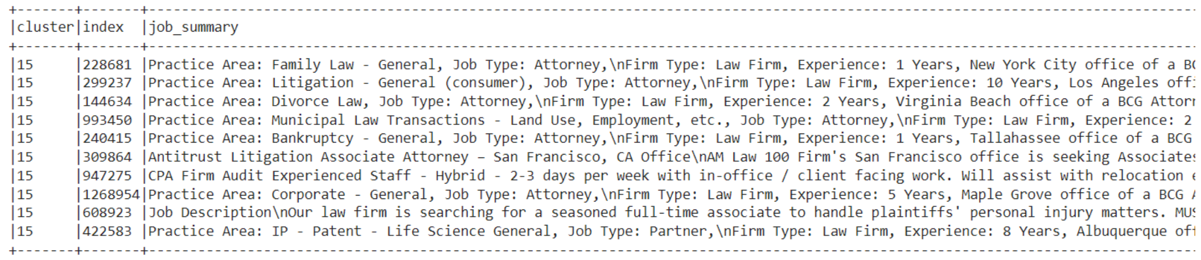
4. Finding sets of similar records

In the final step of the analysis, I employed the Bisecting K-Means clustering algorithm to identify sets of similar records of higher cardinality. Bisecting K-Means is a variant of the traditional K-Means algorithm that uses a hierarchical clustering approach.

The key difference between Bisecting K-Means and regular K-Means lies in the way clusters are formed. Instead of randomly initializing centroids and iteratively assigning points to the nearest centroid, Bisecting K-Means starts with a single cluster containing all data points and then recursively splits clusters into two child clusters until the desired number of clusters is reached. This approach tends to produce more balanced cluster sizes and can handle clusters of varying densities and shapes more effectively.

I used TF-IDF vectors as input to the Bisecting K-Means algorithm and experimented with different numbers of clusters. By analyzing the resulting groups manually, I found that clusters with too many records (100+ records) often contained heterogeneous job postings that were not truly similar. Conversely, smaller clusters (starting from 30-20 records and fewer) tended to contain very similar job postings.

The preference for smaller cluster sizes stems from the nature of the data and the clustering algorithm. Smaller clusters typically exhibit greater homogeneity, containing job postings that share more similar characteristics and keywords. This allows for more precise identification of specific themes or topics within the dataset. Additionally, smaller clusters are more interpretable and easier to analyze, making them preferable for understanding the underlying structure of the data.



```
+-------+-------+--------------------------------------------------------------------------------------------------------------------------
|cluster|index  |job_summary
+-------+-------+--------------------------------------------------------------------------------------------------------------------------
|15     |228681 |Practice Area: Family Law - General, Job Type: Attorney,\nFirm Type: Law Firm, Experience: 1 Years, New York City office of a B
|15     |299237 |Practice Area: Litigation - General (consumer), Job Type: Attorney,\nFirm Type: Law Firm, Experience: 10 Years, Los Angeles off
|15     |144634 |Practice Area: Divorce Law, Job Type: Attorney,\nFirm Type: Law Firm, Experience: 2 Years, Virginia Beach office of a BCG Attor
|15     |993450 |Practice Area: Municipal Law Transactions - Land Use, Employment, etc., Job Type: Attorney,\nFirm Type: Law Firm, Experience: 2
|15     |240415 |Practice Area: Bankruptcy - General, Job Type: Attorney,\nFirm Type: Law Firm, Experience: 1 Years, Tallahassee office of a BCG
|15     |309864 |Antitrust Litigation Associate Attorney – San Francisco, CA Office\nAM Law 100 Firm's San Francisco office is seeking Associate
|15     |947275 |CPA Firm Audit Experienced Staff - Hybrid - 2-3 days per week with in-office / client facing work. Will assist with relocation
|15     |1268954|Practice Area: Corporate - General, Job Type: Attorney,\nFirm Type: Law Firm, Experience: 5 Years, Maple Grove office of a BCG
|15     |608923 |Job Description\nOur law firm is searching for a seasoned full-time associate to handle plaintiffs' personal injury matters. MU
|15     |422583 |Practice Area: IP - Patent - Life Science General, Job Type: Partner,\nFirm Type: Law Firm, Experience: 8 Years, Albuquerque of
+-------+-------+--------------------------------------------------------------------------------------------------------------------------
```

Table 3 - Example of cluster size 30 with Attorney job postings, K-Means 1000

It's important to note that Bisecting K-Means does not allow for direct control over the cluster size; instead, it is a result of the initial definition of *K*, the number of clusters. To ensure the emergence of smaller clusters, K must be chosen large enough to allow for sufficient splitting of clusters. By iteratively adjusting *K* and analysing the resulting clusters, I was able to identify an optimal number of clusters that produced meaningful and interpretable results. I found out that it should be about thousands, e.g. one thousand worked just fine.

5. Conclusions and next steps

In conclusion, this project successfully employed PySpark and various NLP techniques to detect similar pairs and sets of records within a dataset of LinkedIn job summaries. By utilizing Spark's distributed computing capabilities and libraries such as spark-nlp and MLlib, I was able to efficiently preprocess the text data, compute similarity metrics, and perform clustering analysis. Through experimentation, I found that smaller clusters tended to contain more similar job postings, highlighting the importance of cluster size in capturing meaningful patterns in the data.

Moving forward, the next steps could involve further refinement of the preprocessing pipeline, exploration of alternative similarity metrics, and validation of clustering results through domain-specific analysis or manual verification. Additionally, deploying the developed solution on a larger cluster or cloud infrastructure could enable the analysis of the entire dataset and provide deeper insights into the relationships among job postings.