FINDING SIMILAR ITEMS

LinkedIn Jobs & Skills

Magdalena Skowerska

Declaration

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work, and including any code produced using generative AI systems. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

Table of Contents

Abstract

The aim of this project is to find similar pairs or sets of job offers published on Linkedin. The column used to build the similarity detector is job_summary in the job_summary.csv file from the dataset 'Linkedin Jobs & Skills' published on Kaggle under the ODC-By license.
Different techniques will be considered and analized from the performance perspective and the dataset size. TF-IDF technique will be chosen, implemented and analized with the cosine distance measure by using custom functions, sklearn library and pyspark library. Locality Sensitive Hashing (LSH) will be implemented by using datasketch library for small datasets and it will be used in combination with the TF-IDF algorithm for big datasets. Finally, for demonstrative purposes, there will be proposed an implementation of K-Nearest Neighbors (k-NN) suitable for small dataset.

Dataset

The chosen dataset is the ‘Linkedin Jobs & Skills’ for the year 2024, scraped from LinkedIn and published on Kaggle under the ODC-By license. It is composed by three files:

- job_skills.csv containing links to the job offer and the related skills.
- job_summary.csv containing links to the job offer and its description.
- linkedin_job_posting.csv containing job details, company information, locations etc.

The files are in the csv format with tab-separated-values and formatted in the UTF-8 character set.

In this project we will use also the dataset ‘stopwords’ from the python library Natural Language Toolkit (NLTK) which contains stopwords in the English language such as: ‘the’, ‘are’, ‘is’ etc. , and the dataset ‘wordnet’ which groups words into sets of synonyms.

Finding Similar Items Techniques

In order to find similar documents or items in natural language processing (NLP), there are different techniques that can be considered such as:

1. Count vectorization: consisting in transforming a collection of text documents into a matrix where each row corresponds to a document and each columns to the unique token values. The value of the matrix cells represent the count of a specific word in a specific document.
2. Term Frequency times Inverse Document Frequency (TF-IDF): similar to count vectorization method but it adjusts the weights of the words on the basis of their frequency across all the documents.
3. Word Embeddings like Word2Vec, GloVe (Global Vectors for Word Representation) or FastText: represent words as dense vectors in a continuous vector space, capturing semantic relationships.
4. Locality Sensitive Hashing (LSH): based on the usage of the hash function to create buckets of documents that result similar with high probability.

The documents we are dealing with are job descriptions, which can be considered short and of known context, advanced techniques like Word Embeddings will not be considered since they are relatively computationally expensive. Instead, the TF-IDF technique will be implemented that, contrary to the Count vectorization, takes into account the importance of words in relation to a given document making it a more sophisticated approach.

Data preprocessing and Tokenization

The data considered for applying the different techniques comes from the job_summary table consisting of two columns: job_link and job_summary.

As a first step, we apply the data preprocessing techniques including:

- Check and removal of missing values: for both columns there are not null values.
- Check the precence of duplicates: for the column ‘job_link’ the number of distinct values is equal to the number of total rows, so there are not duplicates. For the column ‘job_summary’ the number of distinct value is lower than the number of total rows. In fact, as we can see below (Figure 1) the column contains duplicates. However, they will be kept in the dataset since they are referring to different job links.

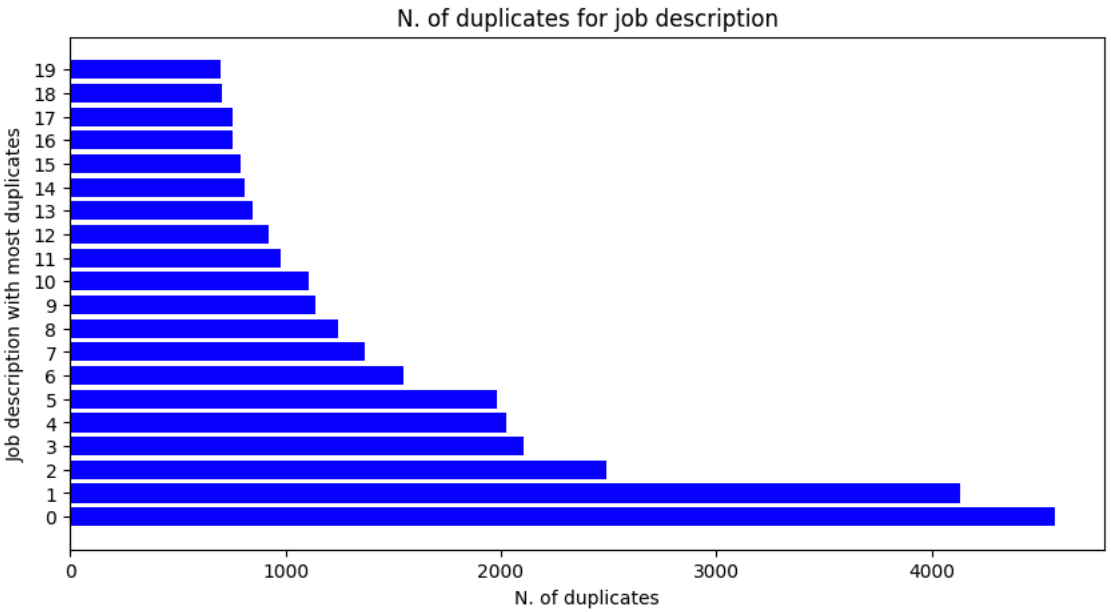


Figure 1 - number of job descriptions duplicates

The second step consists of the tokenization process creating for each job offer a list of words contained in its description. To improve performance and limit the noise in the data we will remove the stopwords from the lists and convert all tokens to lowercase. Furthermore, we will use also the stemming and lemmatizing process considering nltk library:

- Stemming is the process of removing suffixes and prefixes from a word to reduce it to its root form.
- Lemmatization is the process of reducing a word to its base form by considering the sets of synonyms provided by 'wordnet' dataset.

TF- IDF

Term Frequency times Inverse Document Frequency (TF-IDF) consists of calculating:

- **The Term Frequency Index**: given 't' the occurences of a token in a document and 'd' the total number of tokens in a document, the Term Frenquency of the considered token is:

$$TF(t,d) = \frac{t}{d}$$

- **The Inverse Document Frequency**: given a token 't', 'U' the corpus of documents, 'N' the number of documents contained in 'U' and n(t) the number of documents containing the token considered, The Inverse Document Frequency is:

$$IDF(t) = \frac{N}{n(t)}$$

- **TF-IDF** measure of the token in the document considered is:

$$TF(t,d) * IDF(t)$$

First, the algorithm will be implemented by using a custom function. Below (Figure 2 and Figure 3) there are shown two implementations of the function that is used to compute the Inverse Document Frequency values for all of the tokens in the corpus.

```
def idfs(corpus):
    """ Compute IDF
      Args:
          corpus (RDD): input RDD of documents represented by job link and description
      Returns:
          dictionary: dictionary of tokens to IDF values
    """

    unique_tokens = corpus.flatMap(lambda doc: doc[1]).distinct().collect()
    N = corpus.count()
    idfs_val = {token: 0 for token in unique_tokens}

    for token in unique_tokens:
        doc_freq = corpus.filter(lambda doc: token in doc[1]).count()
        if doc_freq > 0:
          idfs_val[token] = N / doc_freq
        else:
          0

    return idfs_val
```

Figure 2 - idf implementation

```
def idfs_optimized(corpus):
    """ Compute IDF
      Args:
          corpus (RDD): input RDD of documents represented by job link and description
      Returns:
          RDD: RDD of tokens to IDF values
    """

    unique_tokens = corpus.flatMap(lambda doc: doc[1]).distinct().collect()
    N = corpus.count()

    token_doc_pairs = corpus.flatMap(lambda doc: [(token, doc[0]) for token in set(doc[1])])
    doc_freq_optimized = token_doc_pairs.map(lambda pair: (pair[0], 1)).reduceByKey(lambda a, b: a + b)
    idfs_val = doc_freq_optimized.map(lambda token_freq_record: (token_freq_record[0], N / token_freq_record[1] if token_freq_record[1] > 0 else 0))

    return idfs_val
```

Figure 3 - optimized idf implementation

When considering a sample dataset of 10 job links, the improvement of the idfs to idfs_optimized function will reduce the computation time from 10 minutes to only 4 seconds. This is done by replacing the filter() and count() functions with flatMap() and reduceByKey(). In fact, the first approach:

- Requires two passes through the data. First, the filter function iterates over all the document to check the presence of that token, then the count function iterates again through the data to count the filtered records.
- Can lead to higher memory usage because the intermediate results of the filtering process must be stored until the count operation is complete.

The second approach on the other hand:

- Generates pairs in a single pass and then aggregates them using MapReduce approach.
- Leads to a lower memory usage because the map() operation creates a smaller representation of the data consisting of tokens and their counts, instead of

having to retain in memory the entire set of documents. ReduceByKey() operation performs counts on the data while processing it on-the-fly.

The optimized version will be used in the tfidf_optimized() function that allows all computation to be done exploiting the RDD data structure suitable for huge amount of data. On the other hand, the function tfidf() is based on the use of dictionaries. So, the limitation of the non optimized function is the number of tokens in a document. Since the tfidf() function returns a dictionary of all tokens and their TF-IDF values per each document, if the number of tokens does not fit in the main memory the application of the algorithm will fail. In our case the documents are short (considering the first 10000 records the maximum length of the job description is 2234) so we can assume that all tokens of a document can fit into the main memory.

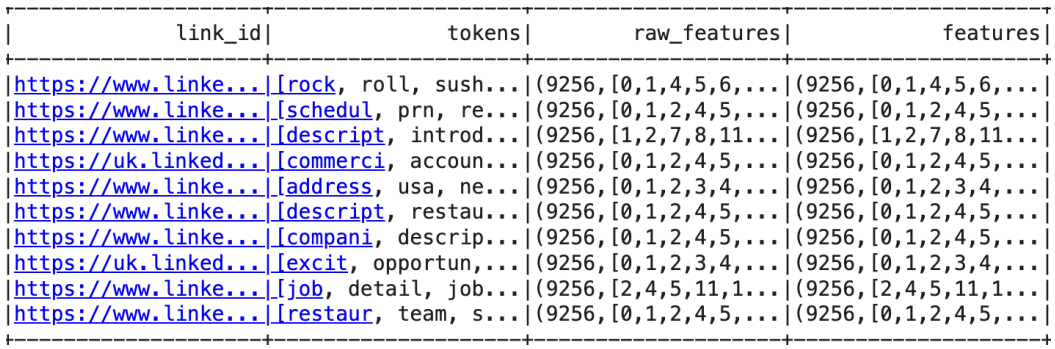
It is possible to use the TF-IDF algortihm with a library scikit-learn built on top of Numpy, SciPy, and matplotlib. This is done by using TfidfVectorizer which is a class that converts a collection of documents into a matrix with one row per document, all the unique tokens as columns and the TF-IDF scores as its values. This class, however, allows in input only documents represented by a list of strings which can be acceptable in case of small datasets but not with larger ones since it brings all data in memory of the driver leading to memory issues. Furthermore, it produces in output a sparse matrix which can result in a not efficient storage usage.

For very large datasets we need to consider the use of the distributed computing system. In this case we can use our custom optimized function or the machine learning library provided by PySpark. In particular, we will use the CountVectorizer class that given all the documents in input produces a matrix with one document per row, the list of tokens and a sparse vector containing the indices of non-zero entries with the relative counts, each representing the number of times the token appears in the document (Figure 4).
The result of the computation will be given as input to the IDF model that calculates the IDF values for each token and produce the TF-IDF vectors for each document. This method is efficient from the memory usage perspective since it stores the sparse matrix as sparse vectors made of (Figure 5):

- the total number of non-zero elements
- the non-zero elements values
- the non-zero elements indices

link_id	tokens	raw_features
https://www.linke...	[rock, roll, sush...	(9256,[0,1,4,5,6,...
https://www.linke...	[schedul, prn, re...	(9256,[0,1,2,4,5,...
https://www.linke...	[descript, introd...	(9256,[1,2,7,8,11...
https://uk.linked...	[commerci, accoun...	(9256,[0,1,2,4,5,...
https://www.linke...	[address, usa, ne...	(9256,[0,1,2,3,4,...
https://www.linke...	[descript, restau...	(9256,[0,1,2,4,5,...
https://www.linke...	[compani, descrip...	(9256,[0,1,2,4,5,...
https://uk.linked...	[excit, opportun,...	(9256,[0,1,2,3,4,...
https://www.linke...	[job, detail, job...	(9256,[2,4,5,11,1...
https://www.linke...	[restaur, team, s...	(9256,[0,1,2,4,5,...

Figure 4 - CountVectorizer output



link_id	tokens	raw_features	features
https://www.linke...	[rock, roll, sush...	(9256,[0,1,4,5,6,...	(9256,[0,1,4,5,6,...
https://www.linke...	[schedul, prn, re...	(9256,[0,1,2,4,5,...	(9256,[0,1,2,4,5,...
https://www.linke...	[descript, introd...	(9256,[1,2,7,8,11...	(9256,[1,2,7,8,11...
https://uk.linked...	[commerci, accoun...	(9256,[0,1,2,4,5,...	(9256,[0,1,2,4,5,...
https://www.linke...	[address, usa, ne...	(9256,[0,1,2,3,4,...	(9256,[0,1,2,3,4,...
https://www.linke...	[descript, restau...	(9256,[0,1,2,4,5,...	(9256,[0,1,2,4,5,...
https://www.linke...	[compani, descrip...	(9256,[0,1,2,4,5,...	(9256,[0,1,2,4,5,...
https://uk.linked...	[excit, opportun,...	(9256,[0,1,2,3,4,...	(9256,[0,1,2,3,4,...
https://www.linke...	[job, detail, job...	(9256,[2,4,5,11,1...	(9256,[2,4,5,11,1...
https://www.linke...	[restaur, team, s...	(9256,[0,1,2,4,5,...	(9256,[0,1,2,4,5,...

Figure 5 - IDF model result

Cosine distance measure

The cosine similarity is based on measuring the cosine of the angle between two vectors in a multi-dimensional space. A smaller angle (closer to the value of 1) means the documents are more similar.
We define our own function that calculate the cosine similarity between two job descriptions given in input an RDD data structure containing job links with the relative dictionary document tokens to its TF-IDF values and a list of the job links for which we want to find similar job descriptions. This function will be used for the optimized version in combination with the custom TF-IDF algorithm. The only memory limitation of this function can be represented by the list of the target job links (for which we want to find similar job description) and the length of the tokenized description. However, in our case this does not represent a problem.
We consider also the use the sklearn implementation when building TF-IDF matrix with the function TfidfVectorizer().fit_transform() that can be used in case of small datasets.

Below we can see a sample of the results obtained by applying the custom implementation (Figure 6) and the implementation by using sklearn (Figure 7).

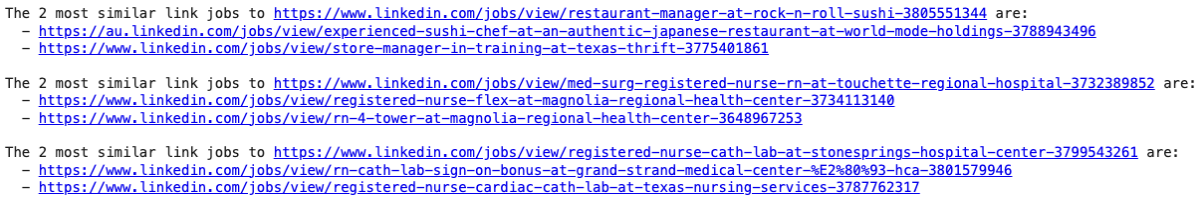


Figure 6 - custom cosine similarity implementation results

Figure 7 - sklearn cosine similarity implementation results

Locality Sensitive Hashing (LSH)

LSH is used for efficient generation of candidate similar items. The algorithm hashes items (like document vectors) into buckets. Items present within the same bucket result similar with high probability, in this way the number of comparisons is reduced to those elements that are in the same bucket.

The similarity measure is done by using Jaccard similarity that compares two sets of tokens extracted from the documents by dividing the size of their intersection to the size of their union.
The project presents the implementation of LSH for a small dataset of 10 records and the implementation necessary in case of huge amount of data using PySpark library. In the second case, the algorithm can be performed on the TF-IDF vectors obtained by applying the PySpark IDF model. To retrieve the most similar items given a link job the approxNearestNeighbors() method will be used based on the cosine similarity between the vectors hashed to the same bucket. The number of hash tables is 10 and for each hash table the number of buckets is set to the default value equal to 128 which is also the number of hash functions (this number can be changed by setting the parameter numHashFunctions). Both MinHashLSH and TF-IDF algorithms from pyspark operate over spark DataFrame built on top of RDD data structure with some optimizations thanks to the Catalyst optimizer and Tungsten engine.

Below we can see a sample of the results obtained by applying the MinHashLSH algorithm for a small dataset (Figure 8) and for large datasets (Figure 9).

Figure 8 - MinHashLSH results for small dataset

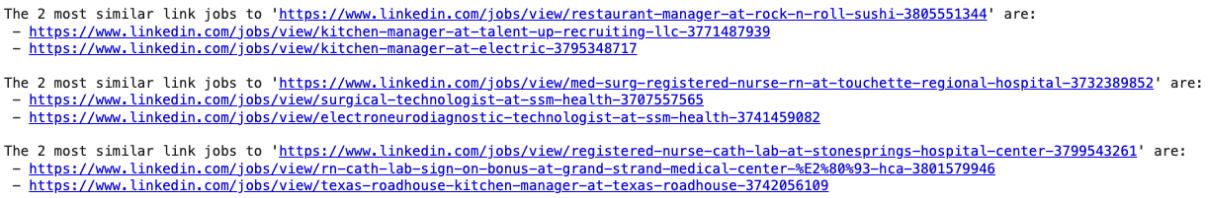


Figure 9 - MinHashLSH results for large datasets

K-Nearest Neighbors (k-NN)

K-nn is a supervised machine learning algorithm based on the fact that similar data points are likely to be close to each other in the feature space and they are grouped in k groups. It uses the cosine distance to measure the distance between points and it identifies the k, in this case equal to two, nearest points (neighbors).

Below we can see the results of the algorithm for our dataset of 10 records (Figure 10).

```
The most similar link job to 'https://www.linkedin.com/jobs/view/restaurant-manager-at-rock-n-roll-sushi-3805551344' is:
- https://www.linkedin.com/jobs/view/restaurant-team-leader-2859-n-germantown-pkwy-germantown-tn-unit-%231300-at-whataburger-3650208124 (Distance: 0.8413)

The most similar link job to 'https://www.linkedin.com/jobs/view/med-surg-registered-nurse-rn-at-touchette-regional-hospital-3732389852' is:
- https://www.linkedin.com/jobs/view/registered-nurse-cath-lab-at-stonesprings-hospital-center-3799543261 (Distance: 0.7846)

The most similar link job to 'https://www.linkedin.com/jobs/view/registered-nurse-cath-lab-at-stonesprings-hospital-center-3799543261' is:
- https://www.linkedin.com/jobs/view/med-surg-registered-nurse-rn-at-touchette-regional-hospital-3732389852 (Distance: 0.7846)

The most similar link job to 'https://uk.linkedin.com/jobs/view/commercial-account-executive-at-the-recruit-lab-3805254225' is:
- https://uk.linkedin.com/jobs/view/concession-store-manager-selfridges-at-linda-farrow-3796128638 (Distance: 0.8478)

The most similar link job to 'https://www.linkedin.com/jobs/view/store-manager-at-stop-shop-3782135496' is:
- https://www.linkedin.com/jobs/view/restaurant-team-leader-3582-highway-114-ft-worth-tx-unit-%231119-at-whataburger-3671804517 (Distance: 0.8582)

The most similar link job to 'https://www.linkedin.com/jobs/view/restaurant-team-leader-3582-highway-114-ft-worth-tx-unit-%231119-at-whataburger-3671804517' is:
- https://www.linkedin.com/jobs/view/restaurant-team-leader-2859-n-germantown-pkwy-germantown-tn-unit-%231300-at-whataburger-3650208124 (Distance: 0.0222)

The most similar link job to 'https://www.linkedin.com/jobs/view/hair-stylist-meadow-brook-center-at-jobs-for-humanity-3786387856' is:
- https://uk.linkedin.com/jobs/view/commercial-account-executive-at-the-recruit-lab-3805254225 (Distance: 0.8743)

The most similar link job to 'https://uk.linkedin.com/jobs/view/concession-store-manager-selfridges-at-linda-farrow-3796128638' is:
- https://uk.linkedin.com/jobs/view/commercial-account-executive-at-the-recruit-lab-3805254225 (Distance: 0.8478)

The most similar link job to 'https://www.linkedin.com/jobs/view/material-handler-at-intelliswift-software-3789502416' is:
- https://www.linkedin.com/jobs/view/restaurant-team-leader-3582-highway-114-ft-worth-tx-unit-%231119-at-whataburger-3671804517 (Distance: 0.9162)

The most similar link job to 'https://www.linkedin.com/jobs/view/restaurant-team-leader-2859-n-germantown-pkwy-germantown-tn-unit-%231300-at-whataburger-3650208124' is:
- https://www.linkedin.com/jobs/view/restaurant-team-leader-3582-highway-114-ft-worth-tx-unit-%231119-at-whataburger-3671804517 (Distance: 0.0222)
```

Figure 10 - k-nn results

Conclusions

This project considers different techniques that could be used to implement an algorithm for finding similar items between set of documents. In our case, the documents are represented by job descriptions and the TF-IDF algorithm was chosen for the aim of the project. More than one way of implementation were proposed for different size of datasets. Among them the ones suitable for big datasets with a large amount of records are: the optimized custom version of TF-IDF with cosine similarity and the implementation of IT-IDF with MinHashLSH using PySpark library. For demonstration purposes, the possibility to use K-nn and MinHashLSH with small datasets was also shown.