Entity Resolution of Publication Data

Kristina Pianykh, Student ID: 617331

*Department of Computer Science, Humboldt University of Berlin,
Unter den Linden 6, Berlin, 10117, Germany.

Corresponding author(s). E-mail(s): pianykhk@hu-berlin.de;

1 Introduction

This report describes the implementation of record linkage, or entity resolution, across the DBLP and ACM datasets containing publication data. This includes cleaning the original data, running the matching algorithm using a similarity function, clustering, and performance analysis. In the end, we provide a quick overview over some techniques for a distributed setup of the pipeline.

1.1 Dependencies

Below are the required specifications to run the code:

- poetry for package management and virtualization (if you do not use it yet, you really should!)
- Python 3.11
- java for running PySpark
- docker
- docker-compose. Note that we use the standalone V1, if this does not work for you, you might want to try running the commands below with `docker compose` if you have the Compose plugin V2.

1.2 Setup

- Make sure you have Python 3.11 installed. If not, you might want to do that with pyenv, undeniably the best Python version installer and manager.
- After installing poetry, make sure to `cd` into the project root and install the Python dependencies of the project:

  ```
  poetry config virtualenvs.in-project true
  poetry install --no-root
  ```

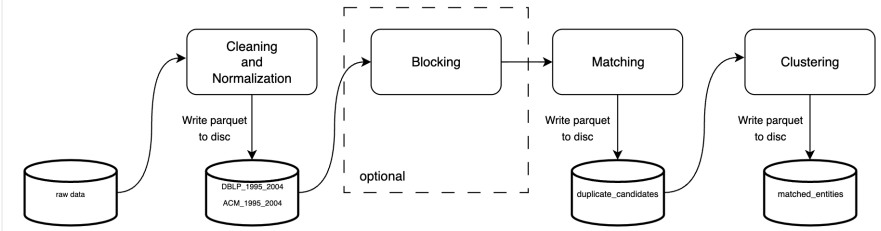
  This will generate `.venv/` directory with all the libraries in the project root.
- Activate the virtual environment with:

```
poetry shell
```

Congratulations! Now you are all set to go!
We will describe each stage of the pipeline separately and provide the commands to run them individually, too. However, if you are impatient, you can run the entire pipeline from the project root as follows:

```
./src/pipeline.sh [--year_range <year_range>]
```

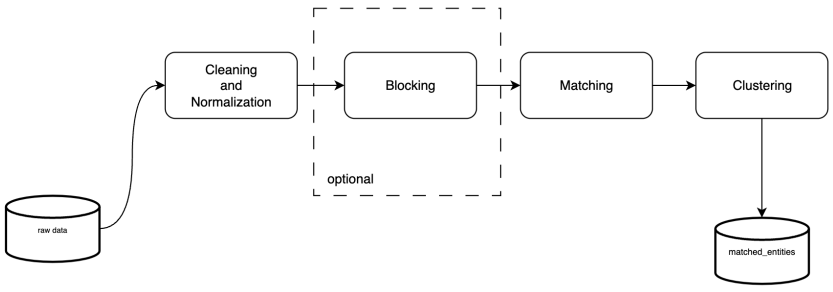
Fig. 1 Pipeline flow as implemented in `src/pipeline.sh`



The script starts with parallel downloading of the archived datasets into the directory `./data/` which will serve as the destination for all the generated data. The download is skipped, however, on every subsequent run. The same counts for downloading the `NLTK` package for the set of English stopwords. The script executes the stages of the pipeline (cleaning, similarity computation, blocking, matching, clustering) as a series of separate Python programs where each of them writes intermediate results in parquet to disc for testing, debugging, and more fine-grained control over each step (Fig. 1). This naturally adds disc I/O overhead and with that inflates the execution time so the pipeline was also re-implemented as a single Python script with data caching to avoid re-computations. It doesn't have any side-effects like the Bash script above and produces the final result `./data/Matched_Entities.csv` only (Fig. 2). Run it from the project root:

```
./src/pipeline.py [--year_range <year_range>]
```

Fig. 2 Pipeline flow as implemented in `src/pipeline.py`



Note that the commands that follow below for each stage of the pipeline are taken from `src/pipeline.sh` so use it for reference.

2 Data Cleaning

The entire pipeline is implemented with PySpark. You can run the cleaning step of the pipeline as shown below by launching the Python script for data cleaning on each dataset in parallel since they are processed independently of each other. Notice that the script expects `./data/citation-acm-v8.txt` and `./data/dblp.txt` as the paths to the raw datasets (this step is covered in `src/pipeline.sh`).

```
python src/prepare_data.py \
    --raw data/citation-acm-v8.txt \
    --dest data/ACM_1995_2004 & \
python src/prepare_data.py \
    --raw data/dblp.txt \
    --dest data/DBLP_1995_2004
wait
```

The first step in the data cleaning process is to load the raw data (originally stored in a txt file) into a tabular data structure. This stage of the pipeline consists of the following steps:

1. We use the double newline separator (`"\n\n"`) to distinguish between the individual publications.
2. We then parse each row (i.e. each publication) in accordance with the original data specifications by extracting the title, the authors, the year, the abstract, the index, and the references based on their respective prefixes (e.g., `#*` precedes the title, `#@` precedes the authors, etc.). The result is a dataframe with the following columns: title, authors, abstract, index, references, year, publication venue. Each value is converted to lower case and trimmed by removing the enclosing whitespaces.
3. Before we proceed with further data transformations, we filter the data by the year (from 1995 through 2004) and publication venue ("SIGMOD" or "VLDB") to avoid expensive regex computations as described below on larger data volumes. The venues are filtered based on whether the strings contain the substrings "sigmod" or "vldp" (note the lowercase).
4. The titles are normalized by removing the stopwords from the `NLTK` library and removing the accents and diacritics to obtain the closest possible representation of the Unicode strings to the ASCII encoding. We do the same thing with the authors but also remove any numbers, multiple whitespaces and special symbols including multiple representations of quotation marks: `!"$%&'()':;+.<=>?‘\"/-@[\\]{|}`. The author names are then sorted alphabetically. The regex rules described above were also applied to the other string columns (except the year, which has the type of an integer in our dataframe scheme).
5. The column with the abstracts was removed because after the filtering step 2. the DBLP dataset has none anyway.
6. We add a column with the number of authors to use later in the matching stage of the pipeline and a column with artificial ID.

3 Similarity Computation and Matching

This stage of the pipeline produces a dataframe with all the pairs of records referring to the same entity. Run this stage of the pipeline on all combinations of the publications as follows:

```
python src/match.py \
    --dblp_path data/DBLP_1995_2004 \
    --acm_path data/ACM_1995_2004 \
    --dest data/duplicate_candidates/full/
```

In this stage, we measure the similarity of publication pairs. At the core of our matching algorithm is a combination of Levenshtein edit distance with Jaccard similarity coefficients.

Once both ACM and DBLP datasets are cleaned, we perform a cross-join (i.e. a cartesian product) to produce one dataframe with all combinations of the records for the next stage of our pipeline. The resulting dataframe has nearly 40 million rows. To gauge the similarity of the records, we relied on the Levenshtein edit distance which measures how many string operations (delete, replace, insert) it takes to bring two strings into an identical form. Since this is a relatively expensive operation to compute, we first filter out all pairs of publications whose venues did not match. In the next step, we compute Levenshtein distance on the strings of authors and discrad the pairs that have a score higher than 9. We then use a number of conditions to further narrow down the scope of the duplicate candidates:

(a) Levenshtein score is 0 and the publications have the same non-0 (non-NULL) number of authors
(b) Levenshtein score is 0 and the publications have 0 (or NULL) authors
(c) Levenshtein score is between 1 and 9 and the publication have the same number of authors

The dataframe filtered based on the above conditions is then used to compute the Jaccard similarity coefficient on the titles of the publication pairs. To do that, we tokenize the titles of each publication and rely on the set semantics to determine the similarity score ranging between 0 (absolute dissimilarity) and 1 (absolute similarity). A brief overview of the results, as well as experiments with different thresholds brought us to the decision to set 0.6 as the final threshold for the Jaccard coefficient. With that, the pairs with Jaccard score below 0.6 are excluded from further consideration.

4 Clustering

This stage of the pipeline is based on grouping the pairs of publications if they all refer to the same entity. It produces a dataframe with a randomly picked pair from the DBLP and ACM datasets as a representative from each cluster. Run the command below to execute clustering, which writes the result in the directory `./data/matched_entities/full/` (or `./data/matched_entities/blocked/` when blocking is used).

```
python src/create_graph.py \
    --duplicates_path data/duplicate_candidates/full/ \
    --raw_dblp data/DBLP_1995_2004 \
    --raw_acm data/ACM_1995_2004 \
    --dest data/matched_entities/
```

The matching technique described in the previous section yields a dataframe consisting of duplicate pairs. However, it would be false to assume that each pair represents a separate cluster, or grouping, in itself. Consider an example of the following duplicate pairs: A-B, B-C, A-D. The pairwise representation does not convey the meaning of transitive property (A-B and B-C $\rightarrow$ A-C) or the relationship "one-to-many" (A-B and A-D), thus failing to group multiple pairs of duplicates into a cluster. We solve this problem by clustering the duplicates using an undirected graph as a data structure. Below is a pseudo-code for building a graph and identifying connected clusters of nodes using a depth-first search algorithm:

```
// create an adjacency matrix
adj_matrix = {}
for row in df:
    dupl1, dupl2 = row
    adj_matrix[dupl1] = dupl2
    adj_matrix[dupl2] = dupl1

graph = []          // list of connected nodes, i.e. clusters
visited = set()     //visited nodes
cluster = []        //cluster of nodes

def dfs(node):
    if (node) in visited:
        return
    visited.add(node)
    cluster.append(node)
    for neighbor in adj_matrix[node]:
        dfs(neighbor)

// populate the graph
for node in adj_matrix.keys():
    dfs(node)
    if cluster:
        graph.append(cluster)
    cluster = []      // reset the cluster
```

This algorithm yields a graph with connected and disconnected clusters of nodes where each cluster represents a group of duplicates. We now pick a random pair from each cluster as a representative and write to the file `./data/matched_entities/full/Matched_Entities.csv` without blocking and to `./data/matched_entities/blocked/Matched_Entities.csv` with blocking. The output file has two columns for the DBLP and ACM

data sets, respectively, and each row represents a pair of records from DBLP and ACM, respectively, mapping to the same publication.

5 Blocking Technique

To utilize a blocking strategy in order to reduce computational workload and execution time when applying the matching algorithm from Section 3, run the command below (replace `<year_range>` with an integer between 1 and 9).

```
python src/match.py \
    --dblp_path data/DBLP_1995_2004 \
    --acm_path data/ACM_1995_2004 \
    --dest data/duplicate_candidates/blocked/ \
    --year_range <year_range>
```

You will notice that the only difference of this command to the one used in Section 3 is the destination path for the output and the parameter `--year_range` that accepts an integer as an input.
Our blocking technique is based on the rolling window of N years and we run the matching step on the datasets filtered by this range. This means that given the sequence of years from 1995 through 2004 and a rolling window of $N = 3$, we filter by the range of 1995 through 1998 in the 1st iteration, then from 1996 through 1999 in the 2nd iteration, then from 1997 through 2000 in the 3rd iteration, and so on. We perform cross-join on the datasets filtered by the year window in each iteration, thus drastically reducing the computation space. The performance metrics of the pipeline with different values for the rolling window are provided in the next section.

6 Matching Performance Metrics

To run the quality metrics, you first have to generate the baseline, i.e. the dataset resulted from running the matching algorithm exhaustively:

```
./src/pipeline.sh
```

Then run the matching using the blocking strategy by a range of `<year_range>` years:

```
./src/pipeline.sh <year_range>
```

The output files are located at `data/duplicate_candidates/full/` and `data/duplicate_candidates/blocked/`, respectively.
Only now can you check how well the blocking of `<year_range>` years has performed:

```
./src/performance.sh
```

6.1 Matching Algorithm Quality

In this section, we measure the quality of our blocking technique described above. The results are provided in Table 1.

Window Size	Precision	Recall	F1	Duplicate count	Baseline Duplicate count
N=1	1.0	0.993	0.996	1592	1604
N=2	1.0	0.999	1.0	1603	1604
N=3	1.0	1.0	1.0	1604	1604
N=4	1.0	1.0	1.0	1604	1604
N=5	1.0	1.0	1.0	1604	1604

Table 1 Performance metrics of the matching and blocking using a rolling window of N years

The algorithm does not misclassify (precision is constant at 1.0) and performs equally well on any year range from $N \geq 2$.

6.2 Execution Time

We measured the execution time that the matching stage of the pipeline takes on the machine with the following specifications:

- Apple M2 Pro Silicon Chip
- 16GB RAM
- macOS Sonoma 14.2.1
- 10 CPU cores

The matching stage of the pipeline was run 5 times for each parameter (the baseline, matching/blocking by the year range of 1, 2, etc.) and the result was averaged over all runs. To run this stage separately and measure the execution time required for generating the baseline, run:

```
time python src/match.py \
    --dblp_path data/DBLP_1995_2004 \
    --acm_path data/ACM_1995_2004 \
    --dest data/duplicate_candidates/full/
```

To run the matching algortihm combined with blocking, run the following (set `<year_range>` to an integer between 1 and 9):

```
time python src/match.py \
    --dblp_path data/DBLP_1995_2004 \
    --acm_path data/ACM_1995_2004 \
    --dest data/duplicate_candidates/blocked/ \
    --year_range <year_range>
```

The measures are provided in Table 2 for both running the matching stage alone, as well as the entire pipeline using the `src/pipeline.sh` and the standalone Python program `src/pipeline.py`.

The results are in line with the assumption that the smaller year window for blocking records into smaller datasets reduces the execution time by reducing the computation space. This observation, combined with the quality estimates from Table1, leads to the conclusion that the optimal speed and accuracy of the blocking technique meet with the year range $N = 2$.

Window Size	Time Matching Stage	Time Pipeline Bash	Python program
baseline	56s	1m16s	52s
N=1	14s	1m	49s
N=2	16s	1m4s	51s
N=3	22s	1m8s	54s
N=4	24s	1m9s	55s
N=5	25s	1m10s	56s

Table 2 Execution time when running the matching with different year ranges N used for blocking

7 Distributed Processing Pipeline

As pointed out in Section 1, we use PySpark to implement the entire pipeline. When used in the local mode, PySpyrk by default utilizes all the existing CPU cores on the machine and as many threads as there are processes available to the Java Virtual Machine. Usually, the number of threads spawned by a PySpark in the local mode equals the number of CPU cores (the master configuration is set to `local[*]` where `*` instructs PySpark to use all the cores that the machine). In other words, on a multi-core machine, PySpark defaults to parallelism, unless explicitly configured otherwise `local[1]`. We used the default setting `local[*]` and ran the pipeline with 10 threads on 10 CPU cores.

To mimic the cluster mode in a local setup, we spun up containerized services using `docker-compose`. It is important to note that we did not use any complex cluster managers like YARN or Mesos other than Spark's native cluster manager. This deployment type is known as the standalone cluster mode and we chose it because of its simplicity.
Start the cluster by running

```
docker-compose up [-d]
```

where the flag `-d` runs the service in the detached mode. Note the IP address of the master service in the docker network to use when submitting a task.
Submit the application task to the master node. Below are the commands to do so to execute the matching on all pairs or subsets by using the blocking strategy by `<year_range>` passed to the parameter `--year_range`:

```
docker-compose exec spark-master spark-submit \
    --master spark://<master-service-IP>:7077 \
    --driver-memory 4000M  \
    --executor-memory 3000M \
    --executor-cores 1 \
      src/pipeline.py [--year_range <year_range>]
```

Note that running Pyspark in a dockerized setup on one machine serves the sole purpose of testing and local development because of its limitations. Virtualization and container isolation come with a cost. While it

provides undeniable benefits in a truly distributed environment with multiple nodes, running a cluster on a single node sets a hard threshold on the capacities such as memory and CPU power since all the containers share the resources of a single node's hardware. Under these circumstances, an increase in execution time is expected considering the overhead of container virtualization.