DIA Exercise Report – Moritz Wassmer

# Description

The executable code is split into two jupyter notebooks. The first one, named *“1\_data\_acquisition\_preperation.ipynb”* for the first task and *“**2\_3\_entity\_resolution.ipynb”* for the second and third task. For the parallel computation framework I chose to use PySpark. The requirements are given in *“requirements.txt”*. The code should run without adjusting any paths. Nethertheless, if there are issues, paths can be adjusted in *“params.py”*.

**Summary:**

Requires

* Spark 3.5
* See requirements.txt

To get the results from report:

1. Set paths in params.py
2. Run 1\_data\_acquisition\_preperation.ipynb
3. Run 2\_3\_entity\_resolution.ipynb

## Data Acquisiton and Preparation

**Loading the data** is achieved by opening the file specified by the path stored in the variable DLBP\_PATH. It opens the file in assuming a UTF-8 encoding and stores all lines in a list.

**Parsing the data** is performed by iterating over each of the lines stored in the list in the following way:

* First of all remove any extra whitespaces
* Then check which of the special patterns applies to map the data entry to a column. Remove the pattern for indication on the type of columns first. Then remove additional whitespaces again.
* If the line is empty, we know the record is finished and a new record begins. The line is added to the result list.
* Then construct a DataFrame from the resulting list.

We then **cast the values** to the respective data type. All string columns are converted to lowercase.

* Year as Integer
* Venue as string
* Index as String
* Title as string
* Authors as string

## Entity Resolution Pipeline

In the following Section I colored the chosen approach from the experiments yellow.

### Blocking

Since I am working alone and wanted to implement an easy to implement, intuitive but flexible solution, I decided to go for the structured keys. I implemented 3 different blocking schemes for comparison and assumed using the similarity measure *‘fancy\_similarity’* described in section ‘Matching’

* **Baseline (Naïve Brutue Force):**
* **Blocking by year:**

Buckets are constructed by the original Year columns.

* **Blocking by year and venue:**  
  Buckets are constructed by the original Year columns with appended substring ‘sigmod’ when there is “sigmod” present or “dldp” when dblp is present.
* **Blocking by authors:**

Buckets are constructed by creating a sorted list without duplicates of characters according to the following scheme: First character of First name, First character of Last name.

For example: “**m**oritz **w**assmer, **f**rederick klaus **m**eier” -> [f, m, w]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Naïve Brutue Force** | **by year** | **by year and venue** | **by authors** |
| **Speed** (minutes:seconds) | 0:29 | 4:43 | 2:05 | 0:08 |
| **TP** | 1460 | 1460 | 1460 | 1446 |
| **FP** | 0 | 0 | 0 | 0 |
| **FN** | 0 | 0 | 0 | 14 |
| **Precision** | 1 | 1 | 1 | 1 |
| **Recall** | 1 | 1 | 1 | 0.9904 |
| **F1** | 1 | 1 | 1 | 0.9952 |

#### Evaluation

Since there is no clear goal defined for this project, I decided to choose for the *bucketization by authors*, since it provides extremely fast runtime while keeping error rates reasonably low. Alternatively to improve the recall, the other blocking schemes could be used.

### Matching

Since we don’t have a ground truth for matching the rows, the only way to compare the quality of different similarity measures is by looking at the matches by hand. Since I can’t go through each record in the report I included a proxy for the strictness of different matching similarities by counting the number of matches.

I first implemented 2 baselines: *simple\_similarit*y and *exact\_match.* After getting an intuition on the behavior of different thresholds and observing the matched results by hand I constructed *fancy\_similarity.*

#### Similarity Measures:

* **Exact Match**

This similarity is simply checking if all attributes are the same in both rows.

* **Simple Similarity**

This similarity measure calculates the distance between each attribute from each dataset and takes an average which is thresholded to determine a match:

It uses levenstein distance for string attributes, and checks wether years in both rows are the same (which is represented by similarity 1, else 0).

* **Fancy Similarity**
* This similarity measure is not thresholded and was constructed in the following way:

Year has to be the same

Authors levenstein distance has to be above 0.9

Title levenstein distance has to be above 0.9

Venue have to both contain substring sigmod or vldb

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Exact Match** | **Simple Similarity** | **Fancy Similarity** |
| **Speed** (minutes:seconds) | 00:07 | 00:09 | 00:09 |
| **# Matches** | 0 | 0.5: 1956  0.6: 1790  0.7: 1621  0.8: 438  0.9: 346  0.5, …, 0.9 are thresholds | 1446 |

#### Evaluation

Since there is no clear goal defined for this project, I decided to choose for the *bucketization by authors*, since it provides extremely fast runtime while keeping error rates reasonably low. Alternatively to improve the recall, the other blocking schemes could be used.

### Clustering

Clustering is achieved by calculating the connected components of the graph resulting from the matched indices and performing fixed point iteration to choose one entity to be kept. I chose not to use a adjacency matrix due to the quadratic size, and went for edge lists as a storage for the graph.

## Data-Parallel Entity Resolution Pipeline

For the data-parallel pipeline I chose to use PySpark.

### Blocking

Similar to the pandas blocking implementation I create a new column named *‘bucket’* according to the *‘Bucketization by authors’* scheme.

### Matching

For matching I do a join and implement the *‘Fancy Similarity’* scheme by performing a join by expression. Additionally, to the expression of the similarity in a string, I add a condition that both *‘bucket’* columns have to match. To implement the blocking scheme I repartition the data according to the Bucket. When the spark engine analyzes the join expression, it should be able to notice that there is no need to compare rows between different nodes since they belong to different buckets. I also implemented another spark pipeline which explicitly filters both dataframes for the same key before performing the join and unioning each individual result. I did not hand in the code for this method, but nethertheless, it was significantly slower.

### Clustering

I decided to reuse the non-parallel clustering function since the runtime of the implementation is very fast (<1 second) and even with big datasets it should be able to store the ids in memory.

### Replication Experiment

For replicating the dataset, a function that randomly changes each attribute according to the following scheme is used:

* For Title, Authors, Venue, and Index, it randomly selects 1 to 3 characters and replaces them with other random characters from the respective column.
* It ensures that spaces in Authors are preserved during the character replacement process.
* For Year, it randomly decides whether to add 0 or 1.

#### Evaluation

Due to the fast blocking scheme, the speed of the data parallel pipeline is almost constant with respect to the replication factor. I also experimented with a blocking scheme which is based on the year, which was significantly slower and speed declined faster for the replication factor.

Ein Bild, das Text, Screenshot, Reihe, Diagramm enthält.

Automatisch generierte Beschreibung