Report for DIA Exercise

- Ali Aslan - 620336
- Silvio Leon Echsle - 381972
- Minh Hieu Le – 492185

The used Python version: 3.10.9

1. Data Acquisition and Preparation: **DIA-ER/dataPreparation.ipynb**

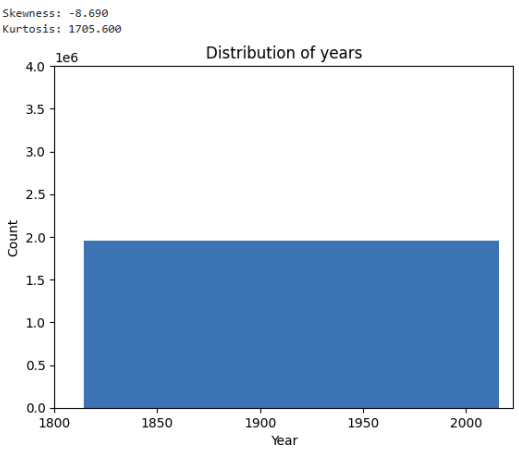
   To be able to execute the data preparation the installation of: gdown, nltk, scipy, Pandas and mumps is required.

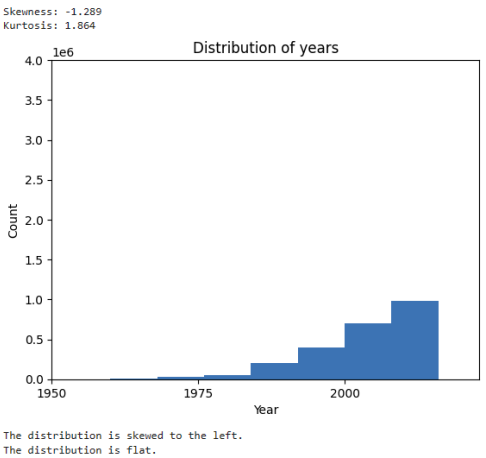
   - The *download_file* function was defined to download the two datasets (*dblp.txt* and *acm.txt*) from Google Drive using the file ID and save them to specified file paths.
   - The *read_file* function reads a custom text file and converts it into a pandas DataFrame, extracting information such as paper title, author names, year, publication venue, and index.
   - The *filling_years* function fills missing values in the 'year'-column of a DataFrame using specified filling methods (median, mean, or mode).

```python
# fill years before filtering
def filling_years(dataframe, filling_method):
    filling_method = filling_method.lower()
    if filling_method not in ['median', 'mean', 'mode']:
        raise ValueError('Invalid filling method')
    if dataframe['year'].isnull().sum().any():
        if(filling_method == 'median'):
            median_year = dataframe['year'].median()
            dataframe['year'] = dataframe['year'].fillna(median_year)
        elif(filling_method == 'mean'):
            mean_year = dataframe['year'].mean()
            dataframe['year'] = dataframe['year'].fillna(mean_year)
        else:
            mode_year = dataframe['year'].mode()
            dataframe['year'] = dataframe['year'].fillna(mode_year)

    return dataframe
```

   - The *distribution_of_year* function creates a histogram and prints statistics (skewness and kurtosis) for the distribution of years in a DataFrame. Below are the outcomes when applying this function to DBLP data and ACM data respectively.





- The *cleansing_data* function fills NaN values for specified columns ('paper_title' and 'author_names') with default values and generates unique identifiers for rows with missing 'index' values.
- The *remove_punctuation_from_columns* function removes punctuation from specified columns in a DataFrame.
- The *clean_text* function applies lowercase transformation, removes leading/trailing whitespaces, and replaces consecutive whitespaces with a single space for text columns.
- The *apply_stem* function applies stemming to text columns using *custom_stemmer* function.
- Missing values in the 'year' column of both *dblp_data* and *acm_data* are filled using the *filling_years* method with mean.
- Only publications between 1995 and 2004 in venues containing “SIGMOD” or “VLDB” are filtered to be shown.
- The *cleansing_data* function is called again to remove duplicates, fill NaN values for 'Authors' and 'Paper', and generate unique IDs for NaN values.
- The punctuations from columns 'paper_title' and 'publication_venue' in both dblp_data and acm_data are removed.
- The *clean_text* function is called to remove leading/trailing whitespaces and replaces consecutive whitespaces with a single space for text columns 'paper_title' and 'publication_venue'.
- Stemming is applied to text columns 'paper_title' and 'publication_venue' using the *apply_stem* function.
- The 'year'-column in both datasets is converted to integer type and reordered.
- Finally, the cleaned and processed data are saved as CSV files in the “CSV-files”-folder.

2. Entity Resolution Pipeline: **DIA-ER/Pipeline/**

   To be able to execute the spark task: install pyspark and alternatively also Hadoop. For Pyspark 3.5.0 it works and if Hadoop is not available and built-in classes of Java are used (works for Java 1.8.0)

   - **blocking_hash.py:**
     - The *initial_hash* function generates blocks by creating a blocking key for each row in the DataFrame based on specified columns, then hashes the blocking key to generate a hash value. At last, the function returns a dictionary where each key is a hash value, and the corresponding value is a list of row indices belonging to that block.
     - The *hash_blocking* function performs hash-based blocking on a DataFrame by concatenating values from specified columns into a hash key, then to generate hash values. Rows with the same hash value are grouped into blocks, and the function returns a dictionary where each key is a hash value, and the value is a list of row indices belonging to that block.
   - **blocking_length.py:**
     - The only function in this file calculates the lengths of specified fields to create a blocking key, then stores the row index in a dictionary with the blocking key as the key and a list of indices as the value. At the end, it returns the dictionary containing the blocking keys and the row indices.
   - **blocking_ngram.py:**
     - The *ngrams_string* function generates n-grams of length *n* from a string *a*.

- The *initial_ngram* function extracts values from specified columns, generates a blocking key using these values, converts the blocking key into n-grams of length *n*, and assigns the record's index to the corresponding block, then returns the blocks
- The *ngrams_tuple* function generates n-grams of length *n* from a given sequence *s* and returns them as tuples.

```
def n_gram_blocking(df, n, columns):
    if 'publication_venue' in columns:
        df ['publication_venue'] = df['publication_venue'].apply(lambda x: 'sigmod' if 'sigmod' in x else 'vldb')

    if 'year' in columns:
        df['year'] = df['year'].astype(str)

    # n-gram for each selected column
    for column in columns:
        key_name = f'ngram_key_{column}'
        df[key_name] = df[column].apply(lambda x: ''.join(map(str, x)))

        ngram_col_name = f'ngram_values_{column}'
        df[ngram_col_name] = df[key_name].apply(lambda x: tuple(ngrams_tuple(x, n)))

    # block them and each block consists of n-gram values and the respective index (id)
    blocks = {}
    for index, row in df.iterrows():
        ngram_values = {column: row[f'ngram_values_{column}'] for column in columns}
        hashable_ngram_values = hashlib.sha256(str(hash(tuple(sorted(ngram_values.items())))).encode()).hexdigest()

        if hashable_ngram_values in blocks:
            blocks[hashable_ngram_values]['index'].add(row['index'])
        else:
            blocks[hashable_ngram_values] = {'index': {row['index']}, **ngram_values}

    return blocks
```

- The *n_gram_blocking* function converts each selected column's values into n-grams, then hashes and sorts these n_grams along with their respective column names to create blocking keys. Finally, it assigns record indices to the blocks based on the blocking keys. The resulting blocks are returned as a dictionary.

- **blocking_structured_and_sort.py:**
  - The *block_by_year_ranges* function divides the data into blocks based on specified year ranges, returns a dictionary where each label corresponds to a list of intervals containing records falling within the specified year range.
  - The *block_by_year_and_publisher* function divides data into blocks based on both year ranges and publishers additionally, returns a list of intervals containing records falling within each year range and associated with specific publishers.
- **cluster.py:**
  - The *build_clusters* function goes through each matched entity pair and checks if either entity is already in an existing cluster. If that is not the case, it creates a new cluster. If one of them is found in multiple clusters, it merges those clusters and returns them.

- The *clustering_matches* function also goes through each matched entity pair but directly merges the entities into clusters and if any of their values are in an existing cluster. Otherwise, it creates a new cluster. The list of clusters is returned at the end.

- **matchers.py:**
  - The *apply_similarity_baseline* function compares entities from two dataframes based on selected columns using a specified similarity function, then returns pairs of entities with similarity scores above a given threshold.

```
def apply_similarity_baseline(df1, df2, threshold, selected_columns, similarity_function):
    similar_pairs = []

    for index1, row1 in df1.iterrows():
        for index2, row2 in df2.iterrows():
            average_similarity = 0.0

            for col_df1, col_df2 in zip(selected_columns, selected_columns):
                value_df1 = set(row1[col_df1].lower().split())
                value_df2 = set(row2[col_df2].lower().split())

                similarity = similarity_function(value_df1, value_df2)
                average_similarity += similarity

            if len(selected_columns) > 1:
                average_similarity /= len(selected_columns)
            if average_similarity >= threshold:

                pair = (row1['index'], row2['index'])
                similar_pairs.append(pair)
    return similar_pairs
indices = ['author_names', 'paper_title']
```

  - The *apply_similarity_blocks* function compares entities within blocks generated by blocking techniques, and computes similarity scores between entities in each block pair and returns pairs with similarity scores above the threshold.
  - The *apply_similarity_sorted_dictionary* function compares entities from sorted blocks represented as dictionaries, matches entities within corresponding blocks and returns pairs with similarity scores above the threshold.
  - The *apply_similarity_lengths* function compares entities based on their lengths, returns pairs of entities with lengths above a given threshold.

```python
def apply_similarity_lengths(blocks1, blocks2, threshold, similarity_function, indices):
    similar_pairs = []

    # Record the start time
    start_time = time.time()

    for key1, block1 in blocks1.items():
        for key2, block2 in blocks2.items():
            average_similarity = 0.0

            value_block1 = [block1.get(indices)]
            value_block2 = [block2.get(indices)]
            similarity = similarity_function(value_block1, value_block2)

            if similarity >= threshold:
                index_pairs = [(i, j) for i in block1['index'] for j in block2['index']]
                similar_pairs.extend(index_pairs)

    # Record the end time
    end_time = time.time()

    # Calculate the elapsed time
    elapsed_time = end_time - start_time

    # Print the elapsed time
    print(f"Processing time: {elapsed_time} seconds. Number of similar pairs: {len(similar_pairs)}")

    return similar_pairs
```

- **similarity.py:**
    - The *levenshtein_distance* function calculates the Levenshtein distance between two sets of elements, representing the minimum number of single-character edits required to change one string into the other. It then returns the similarity score as 1 minus the normalized Levenshtein distance.
    - The *jaccard_similarity* function computes the Jaccard similarity coefficient between two sets, representing the size of the intersection divided by the size of the union of the sets. The function then returns a similarity score between 0 and 1, where 0 indicates no similarity and 1 indicates identical sets.
    - The *jaccard_similarity* function determines the Jaccard similarity for sets of N-grams, which are contiguous sequences of n items from a longer set, and returns a similarity score between 0 and 1 to indicate the overlap of sets.
    - The *n_gram_similarity* function computes the intersection over the combined size of the sets of N-grams and returns a similarity score indicating the overlap between sets.
    - The *n_gram_similarity2* function also measures the intersection over the combined size of the sets and returns a similarity score.
    - The *exact_length_similarity* function simply compares the lengths of two sets and returns a binary similarity score saying whether the lengths are the same or not.
- **entity_resolution_pipeline.ipynb:**
    - First of all, the prepared datasets *dplp_stem.csv* and *acm_stem.csv* are read.

- **o** The s*imilar_pairs_to_csv* function is defined to write pairs of similar entities to a CSV file. It takes a list of similar pairs and an output CSV file path as input.
- **o** The *evaluate_similarity* function was written to evaluate the similarity between two sets of pairs and calculate precision, recall, and F-measure.
- **o** The 'year'-column in both datasets is converted to strings.
- o Baselines: For the Baseline we matched our read data without blocking and used the similarity functions: Jaccard, N-gram, exact length, Levehstein distance, with the respective thresholds 0.7 and 0.85 and saved them in the "baselines"-folder
- **o** The csv-files are transformed into a list of pairs using the *reconstructed_pairs* function.
- **o** The *blocks_to_df* function transforms the created blocks from the sorted blocking (the one with year and publisher) into dataframes again so we can apply on these dataframes a second blocking technique which enables the use of multiple blocking techniques I sequence.
- **o** The *block_dfs* function organizes DataFrames into blocks using a given blocking function, so that we can apply multiple blocking techniques for enhanced entity resolution.
- o Then, different blocking methods such as hash blocking, n-gram blocking, and length blocking are applied on two datasets DBLP and ACM, before the similarity scores between pairs of entities are determined using various similarity functions liek Jaccard similarity for n-grams.

- o The next 4 code blocks evaluate the performance of different sorted blocking techniques by comparing their similarity results with a baseline Jaccard measure. Precision, recall and F-measure are computed for different sorted blocking strategies across various data attribute subsets.

- o The following code snippets perform various blocking methods such as n-gram, hash, length, and sorted blocking on DBLP and ACM datasets, using different combinations of attributes. The Jaccard similarities between blocked records are calculated based on thresholds and attribute sets.

- o The next 4 code blocks evaluate the similarity between 'base_85_jac' and different sets of attributes sorted in various ways.

- o Thereafter, length-based blocking is performed on different combinations of attributes from both datasets. The *apply_similarity_length* function computes the similarity between corresponding attribute values with a threshold of 0.7. The results are stored in variables prefixed with 'length_'.

- o Subsequently, the similarity between two datasets is measured based on length similarity for combinations of attributes.

```
result_combined_length = (
    evaluate_similarity(base_7_l, length_a) +  "\n" +
    evaluate_similarity(base_7_l, length_p) +  "\n" +
    evaluate_similarity(base_7_l, length_ap) +  "\n" +
    evaluate_similarity(base_7_l, length_apv) +  "\n" +
    evaluate_similarity(base_7_l, length_ay) +  "\n" +
    evaluate_similarity(base_7_l, length_appv) +  "\n" +
    evaluate_similarity(base_7_l, length_apy) +  "\n" +
    evaluate_similarity(base_7_l, length_appvy) +  "\n" +
    evaluate_similarity(base_7_l, length_ppv) +  "\n" +
    evaluate_similarity(base_7_l, length_ppvy)
)
```

- The *read_matched_entities* function reads matched entities from a CSV file, skipping the header. Each cluster and its index are then printed by the *print_clusters* function. Finally, the *cluster_tocsv* function writes the clusters to a new CSV file.
- The chosen matched entities was for the baseline with jaccard similarity and a threshold of 70%. The succeeding blocking technique was initial_hash with the usage of the attribute 'paper_title'. The measures for precision, recall and f-measure are: 0.90, 0.88 and 0.89.

3. Data-Parellel Entity Resolution Pipeline: **DIA-ER/spark**
   - **blocking_structured_and_sorted_spark.py**
     - The only function uses PySpark to perform data blocking based on year and publisher, then divides the data into blocks based on year ranges and publisher labels, and finally returns a list of lists containing data records.
   - **cluster_spark.py**
     - The defined function converts data to a DataFrame, then merges related clusters to construct new ones.
   - **hash_blocking_spark.py**
     - The *hash_partition* function computes the hash value for each blocking key, and yields a tuple containing the blocking key, hash value and associated indices.
     - The *initial_hash_parallel* function creates a blocking key column by merging transformed values, computes hash values for each key, and returns a DataFrame.
     - The *initial_hash_parallel_df* function transforms specified columns, generates blocking keys, groups data by these keys, and creates a DataFrame.
     - The *hash_blocking_spark* function creates hash keys from specified columns, groups data by these keys, and aggregates indices into lists. The resulting blocks are collected and converted into a DataFrame.
   - **length_blocking_spark.py**
     - The *length_blocking_parallel* function defines two functions *udf_length_int* and *udf_length_str* to calculate lengths of values, then adds a new column "value" with the lengths of values in specified columns, and finally constructs a blocking key by concatenating lengths of values in specified columns with underscores as separators.
   - **matchers_spark.py**
     - The *blocking_by_year_and_publisher_column* function creates a blocking key column based on the year and publication venue columns of the DataFrame.
     - The *apply_similarity_sorted* function applies a similarity function to pairs of blocks from two DataFrames and returns pairs of indices that meet the similarity threshold.

- The *apply_similarity_blocks_spark* function applies a similarity function to pairs of blocks from two DataFrames using Spark, and returns pairs of indices that meet the similarity threshold.
- The *apply_ngram_blocks_spark* function is similar but designed for n-gram blocking.

- **ngram_blocking_spark.py**
  - The *ngram_partition* function generates n-grams from the blocking keys in each partition of the DatFrame.
  - The *initial_ngram_parallel_df* function creates blocking keys, generates n-grams for each and aggregates them.
  - The *n_gram_blocking* function generates n-grams for each specified column, concatenates the n-grams, creates blocking keys and aggregates the data.
- **similarity_spark.py**
  - The functions compute various similarity metrics between sets of data.
- **er_spark.ipynb**
  - The *write_index_pairs_to_csv* function cleans each row by removing brackets, and writes the cleaned rows along with a header to the specified CSV file.
  - The *reconstructed_pairs* function reads the CSV file, retrieves the 'dblp_index' and 'acm_index' columns, and zips them into a list of tuples (for the matched entities in 2).
  - The *pairs_correspond* function sorts the input lists of pairs and checks if each pair is present in the other list. If that's the case, then True is returned, otherwise False.

```
# check if they are equal
def pairs_correspond(pair_list1, pair_list2):
    sorted_pair_list1 = [tuple(sorted(pair)) for pair in pair_list1]
    sorted_pair_list2 = [tuple(sorted(pair)) for pair in pair_list2]

    for pair1 in sorted_pair_list1:
        if pair1 not in sorted_pair_list2:
            return False
    return True

pairs_correspond(pairs_spark, pairs)
```

    This ensures that the results of the matched entities are identical with the corresponding data parallel function.

  - The *cluster_to_csv* function is defined to cluster the index pairs within the list and writes the solution into an CSV file.
  - The next code snippet applies different combinations of augmentation functions to the 'paper_title' of ACM and DBLP datasets, then computes matches using initial_hash_parallel_df and the jaccard similarity with a threshold of 70%, and finally prints the results of each combination.
  - The last code block reads ACM and DBLP datasets using Spark, applies n-gram blocking on selected columns, computes matches and stores the results. This shows that the other implemented data parallel functions work and it serves as an example.