Assignment 2

Nisse Hermsen, Isabelle de Wolf, Sara Sakhi,
Celis Tittse, Agnieszka Kubica

October 6, 2024

Introduction

This report presents a variety of data preparation techniques. Nisse Hermsen, Isabelle de Wolf, Sara Sakhi, Celis Tittse, and Agnieszka Kubica collaborated on this assignment.

Task 1: Profiling relational data

This assignment describes eleven different summary statistics on a dataset of Latin excerpts from the Library of Latin Texts (Brepols). The dataset investigated the antisemitic rhetoric in the Late Antique Latin world (2nd to 8th century AD). It consists of excerpts (a small portion of text) of digitised texts that mention versions of the words synagogue (synagogue), ecclesia (church), and domus (house). These excerpts were tokenized with lemmatisation and the removal of Latin stopwords. A topic modelling was performed with seven topics, and a sentiment analysis was performed on the tokenized excerpts. The extraction of the Library of Latin Texts and the results of these analyses will be interrogated with summary statistics in this assignment.

The summary statistics and more complex statistics are taken from the paper *Profiling relational data: a survey* by Abedjan, Golab, and Naumann (2014).

```
# loading the dataset
excs_df = pd.read_excel('./data/excerpt_topics_total_7.xlsx')

# explore the head and tail of the dataframe to check if the dataset is loaded correctly
print(excs_df.head)
print(excs_df.columns)
```

We want to grasp the size of the data, in this case, the number of Latin excerpts that we are working with. To this end, we deployed the number of rows of the ID column, which should be this table's primary key. To check this, we divided the number of rows by the number of distinct ID column values. In addition, the uniqueness of the raw text will be examined to cover overlapping excerpts.

```python
def primary_key_check (df: pd.DataFrame):
    id_col = df['ID']
    text_col = df['text']

    len_id = len(id_col)
    dist_id_len = len(id_col.unique())
    dist_text_len = len(text_col.unique())

    # unique text n / unique ID n
    if dist_text_len / dist_id_len == 1:
        print(f'The unique texts of the excerpts as long as the unique IDs')
    else:
        print(f'The text are not unique:  {dist_text_len}/ {dist_id_len}')

    # unique ID's / total ID's
    if dist_id_len / len_id == 1:
        print(f'ID is a primary key \n'
              f'and the number of excerpts is {len_id}')
    else:
        print(f'ID is not a primary key \n'
              f'and the number of unique excerpts_ids is {dist_id_len}')
```

Now we know that the ID is the primary key and the text is a candidate key, and thus, we work with unique excerpts.

We wanted to know the minimal and maximal length of words to better understand the size of the text we were working with. To be more precise, we created a boxplot of the number of words in all excerpts. This lets us know how large the snippets of texts we're working with.

```python
def boxplot_words_excs (texts: pd.Series):
    # count the number of elements distinguished by spaces in the excerpts
    word_count = [len(exc.split(' ')) for exc in texts]

    print(f'The range of words are between {np.min(word_count)} and {np.max(word_count)}\n'
          f'with an average of {round(np.mean(word_count), 4)} and\n'
          f'with 95% of the excerpts between {np.percentile(word_count, q= [2.5, 97.5])}')

    plt.boxplot(word_count)
    plt.ylabel('Word (n)')
    plt.title('Boxplot of the words per excerpts')
    plt.show()
```

The number of words within the excerpts can differ intensely (4 to 1093 words). However, 95% of the data remains between 24 and 180 words. Thus, we work with small sentences and small paragraphs.

The dataset has two essential attributes that can be utilised in analysis: The century in which an excerpt was probably written and the religious keyword

used. If the dataset is to be analysed as a whole, it is necessary to know its representativity in different centuries. It is also essential to know how many excerpts are anonymously written. If a temporal analysis is conducted, we should decide what to do with these data points. Furthermore, it is necessary to investigate if multiple keywords are not covered in one excerpt entry. The main concern of this dataset is the difference in excerpts with these keywords. If keywords would overlap, it could influence the results.

```
def histo_cent (cent: pd.Series):
    print(f'Number of NAs in century: {cent.isna().sum()}')

    # histogram of the centuries with a bin for every century
    plt.hist(cent, bins = len(cent.unique()))
    plt.xlabel('Century')
    plt.ylabel('Excerpts (n)')
    plt.title('Histogram of excerpts in Late Antiquity')
    plt.show()
```

The unknown authors are dated or excluded from the dataset. The authors of these excerpts mostly wrote in the 4th to the 6th century. This could resemble an imbalance in the digitization process or data gathering, the dominance of Greek in the 2nd and 3rd centuries in intellectual circles, and a steep decrease in Latin writing after the collapse of the administration of Rome in 476.

```
def keyword_labeling (df):
    # slice df to lemmitized tokens of the text and the religious keywords
    token_kw_df = df[['tokens', 'keyword']]

    # get unique keywords
    uniq_kws = token_kw_df['keyword'].unique()

    search_tokens = ['synagoga', 'ecclesia', 'domus']

    results = []
    for kw in uniq_kws:
        # get subsets of the tokens with the unique keywords
        sub_tokens = token_kw_df[token_kw_df['keyword'] == kw]['tokens'].tolist()

        sub_result = []
        for s_tok in search_tokens:
            # if the religious word is in tokens than add 1 else 0 and sum the result
            sum_search_words = np.sum(1 if s_tok in token else 0 for token in sub_tokens)
            sub_result.append(sum_search_words)

        results.append(sub_result)

    # print the dataframe of the results
```

```
    results_df = pd.DataFrame(results, columns=search_tokens, index=uniq_kws)
    print(results_df.to_string())
```

As labelled in the data, the keywords are not exclusionary for the other keywords. They are almost exclusionary, but not entirely. For further analysis, we should consider the overlapping excerpts.

So far, we have interrogated the structure of the data, the uniqueness and size of the excerpts, and the distributions and quality of the attributes keywords and centuries. The last phase of the EDA concerns the Sentiment Analysis (SA). The SA performed in this dataset was a static SA, which considers a labelled list of Latin words with an emotional connotation. The sum of the SA-labelled tokens per excerpt was noted under the SA score.

1. Does it look like data from the real world? To determine this, we will use Benford's Law. This will quickly check if the scores conform to Berford's law. 2. the SA score assumes that it remains similar over the centuries. This can be tested with a scatter plot and a regression line.

```
def berford_sa_score(sa: pd.Series):
    # Convert values to string, remove the negative sign, and select the first character
    leading_digits = [int(str(abs(score))[0]) for score in sa.dropna()
        if isinstance(score, float)]

    # Compute the actual frequencies of leading digits
    actual_count = Counter(leading_digits)
    total_count = sum(actual_count.values())  # Total number of valid leading digits
    # Normalize to proportions
    actual_freq = [actual_count[d] / total_count for d in range(1, 10)]

    # Compute the expected frequencies according to Benford's Law
    benford_freq = [np.log10(1 + 1 / d) for d in range(1, 10)]

    # Compare actual vs expected (Plotting)
    digits = range(1, 10)
    plt.figure(figsize=(10, 6))
    plt.bar(digits, actual_freq, width=0.4, label='Actual', align='center', alpha=0.7)
    plt.bar(digits, benford_freq, width=0.4, label='Benford', align='edge', alpha=0.7)
    plt.xlabel('Leading Digit')
    plt.ylabel('Proportion')
    plt.title('Comparison of Leading Digit Distribution with Benford\'s Law')
    plt.xticks(digits)
    plt.legend()
    plt.show()
```

Benfordś Law remains preserved. The distribution of first digits is proportional to the log of 1 + 1/digit. This means that the distribution of this SA score has a probability of serious errors.

```python
def scatter_with_regression(df):
    # remove na's
    clean_df = df[['century', 'sa_score']].dropna()
    cent = clean_df['century']
    sa_score = clean_df['sa_score']

    # Create a scatter plot
    plt.figure(figsize=(10, 6))
    plt.scatter(cent, sa_score, color='blue', label='Data Points', alpha=0.2)

    # Fit a regression line
    slope, intercept, r_value, p_value, std_err = scipy.stats.linregress(cent, sa_score)

    # Regression line
    x_vals = np.array(cent)
    y_vals = intercept + slope * x_vals
    plt.plot(x_vals, y_vals, color='red', label=f'Regression Line (slope={slope:.2f})')

    # Annotating the slope on the plot
    plt.text(x=cent.mean(), y=sa_score.min(), s=f'Slope: {slope:.2f}',
        color='red', fontsize=12)

    # Labels and title
    plt.xlabel('Century')
    plt.ylabel('SA Score')
    plt.title('Scatter Plot of SA Score vs Century with Regression Line')
    plt.legend()

    # Show plot
    plt.show()

    print(f'Slope: {slope:.2f}')
```

A temporal analysis of the excerpts' sentiments shows an increase in the total sentiment over the century (r=0.28). This means that a normalisation step needs to be performed on the SA when other attributes like dominant topics or keywords are considered. The total number of words with a positive connotation is used in later centuries compared to earlier ones. This might be a historical reason that needs further investigation.

The dataset of Latin excerpts from the Library of Latin Texts was explored using ten summary statistics. The primary key check confirmed the uniqueness of the excerpts, and the word length analysis revealed that most excerpts are small passages, with 95% containing between 24 and 180 words. The dataset is imbalanced across centuries, with most excerpts from the 4th to 6th centuries likely reflecting historical factors. The keywords "synagogue," "ecclesia," and "domus" show minimal overlap, preserving the distinctiveness of excerpts by

keyword. Benford's Law confirmed the validity of the sentiment analysis (SA) scores. Still, a temporal analysis revealed a positive trend in SA scores over time, suggesting a potential bias in sentiment that needs to be accounted for in further analysis.

Task 2: Entity resolution

Part 1

All of the mentioned code and their components are called in a wrapper function called `main`. This method is responsible for the order of execution and printing the results:

```
def main():
    df_acm, df_dblp, df_mapping = load_data()

    # Preprocess text.
    df_acm = preprocess(df_acm)
    df_dblp = preprocess(df_dblp)

    # Find similar records.
    time_start = time.time()
    df_sims = pairwise_comparison(df_dblp, df_acm)
    duration = round(time.time() - time_start, 2)

    # Filter similar records.
    df_sims = df_sims[df_sims['sim_score'] > 0.7]

    # Calculate precision
    df_match = join_mapping(df_sims, df_mapping)
    n_matches = df_match['match'].sum()
    precision = round(n_matches / len(df_match), 2)

    # Print results.
    print(df_match.head())
    print(df_match.describe())
    print(df_match.info())
    print(f"Precision is {precision} with {n_matches} correct matches.")

    # Runtime
    print(f"Runtime of the pairwise similarity comparison is {duration} seconds.")
```

a. Ignore the id column.

The `id` attribute in both the `ACM` and `DBLP` dataset is ignored by converting it to be an index. We opt for this solution, rather than removing it from the

dataframe, because we want to use that attribute to identify the duplicated data later on in point j. The conversion was made using the following function applied on both datasets:

```
df = df.set_index('id')
```

b. Change all alphabetical characters into lowercase.

To perform this point and point c. we defined the function:

```
def preprocess(df: pd.DataFrame) -> pd.DataFrame:
    """
    Set the id columns as the index and preprocess text columns. Preprocessing includes
    removing whitespaces, converting to lowercase and replacing missing values with the
    empty word "".
    :param df: Dataframe containing raw data
    :return: Dataframe with preprocessed data
    """
    df = df.set_index('id')

    for col in df.columns:
        if df.dtypes[col] != 'object':
            continue

        # Lowercase
        df[col] = df[col].str.lower()

        # Whitespaces
        df[col] = df[col].str.replace(r'\s+', ' ')

        # NA values
        df[col] = df[col].fillna("")

    return df
```

The function was then applied on both datasets. It loops over the columns of a dataframe and verifies if the attribute is an object. This verification is applied to remove columns that include numerical data from the string preprocessing. Then, the .lower() function is applied on all of the string elements of the array.

c. Convert multiple spaces to one.

The string preprocessing function (included in b.) first converts all elements object type columns to lowercase. Then, .replace(r'\s+', ' ') is applied on all elements of the array. 'r' informs the programm that the expression that follows it is a regular expression and not just a string. The'\s+' expression finds all

white space characters in each row, this includes multiple spaces, but also new lines and tabs. The are replaced by a single space.

d. Compute Levenshtein similarity on title attribute.

Initially, we solved this exercise by implementing a self written algorithm that computed the Levenshtein similarity based on the implementation in the labs. However, we quickly learned that this algorithm is far from optimised. Computing a matrix for every one of the comparisons highly inflated the running time of the program. Instead, we opted for an in built function from the 'Levenshtein().get_sim_score()' from the py_stringmatching package. The Levenshtein similarity alongside other similarity measures were implemented by the code and function below, later applied to the two datasets:

```
    # Initiate models.
    lev = sm.Levenshtein()
    jaro = sm.Jaro()
    aff = sm.Affine()

def record_sim(r: pd.Series, lev: sm.Levenshtein, jaro: sm.Jaro, aff: sm.Affine)
    -> pd.Series:
    """
    Apply py_stringmatching built-in similarity scores to a single record. Note that venue
    (using the affine similarity) is not normalised.
    :param r: data record
    :param lev: Levenshtein class instance
    :param jaro: Jaro class instance
    :param aff: Affine class instance
    :return: Record including similarity scores per attribute
    """
    r['sim_title'] = lev.get_sim_score(r['title_x'], r['title_y'])
    r['sim_authors'] = jaro.get_sim_score(r['authors_x'], r['authors_y'])
    r['sim_venue'] = aff.get_raw_score(r['venue_x'], r['venue_y'])
    r['sim_year'] = year_sim(r['year_x'], r['year_y'])
    return r
```

The models were initiated separately, since we reuse the class instance for performance. We want to avoid re-initialising the model at each record iteration.

e. Compute Jaro similarity on authors attribute.

Similarly as in part d, we initially wanted to implement code similar to the lab exercises, but in the end we opted for a more optimised function Jaro().get_sim_score() from the py_stringmatching package. The similarity was computed on the cross joined authors columns (see code in part d.).

f. Compute affine similarity on venue attribute and scale it to [0,1].

Analogically to points d. and e., the similarity was implemented on the venue attribute using Aff().get_raw_score() from the py_stringmatching package. We opted for the default cost values: 1 for a gap to start and 0.5 for continuation. The range of the affine similarity score was later re-scaled to the [0,1] interval (see i.) using the following code:

```
# Apply min-max scaling (normalisation) to sim_venue.
df['sim_venue'] = (df['sim_venue'] -
df['sim_venue'].min()) / (df['sim_venue'].max() -
df['sim_venue'].min())
```

From each value of affine similarity of venues the minimum value was substracted, making the new minimum 0. Then, the new values were divided by the range of original values, leading the maximum value to reduce to 1.

g. Use Match (1) / Mismatch (0) for the year.

Similarity of year was understood as: the year is the same, thus the records are similar and take value 1, or the year is not the same and there is no similarity in the year value, therefore it takes value 0. This similarity was computed using the following if function:

```
def year_sim(s, t) -> int:
    """
    Match / mismatch between two year instances. Returns 1 if matched.
    :param s: Year instance left
    :param t: Year instance right
    :return: Match result
    """
    if s == t:
        return 1

    return 0
```

Then it was implemented on the year attributes of the datasets, as can be seen in code in part d.

h. Convert the similarity scores into one index - rec_sim.

Once we computed the similarity measures between the different attributes, we created an overall similarity score between a pair of records: `sim_score`. It was computed by the following part of the function presented at the beginning of exercise 2:

```
# Calculate similarity score.
df['sim_score'] = 0.45 * df['sim_title'] + 0.45 * df['sim_authors'] + 0.05 *
```

```
        df['sim_venue'] + 0.05 * df['sim_year']
    return df
```

We opted for weights 0.45 for title and author similarity and 0.05 for venue and year. This decision was based on the uniqueness of values of each attribute. The amount of unique entries in both datasets within each attribute were computed using the `nunique()` function and are presented in Table 1.

Table 1: Number of unique entries

Attribute	ACM	DBLP
title	2230	2521
authors	2008	2316
venue	5	5
year	10	10

Attributes `title` and `authors` have drastically more unique values than the other two and the uniqueness of `venue` and `year` is minimal compared to the total number of entries. Because of that, we concluded that the first two attributes are far more informative in the process of finding similar or even duplicate records. Therefore, we assigned a much larger weight to them.

i. Report the records with rec_sim $>$ 0.7 as duplicate records by storing the ids of both records in a list.

The above implementations calculate the attribute specific similarity scores *per* record. To summarise each component score into a final score called `sim_score`, each component must be normalised to a range of $[0, 1]$. In our case, the `sim_title`, `sim_authors` and `sim_year` attributes were already normalised as per the use of the built-in `py_stringmatching` functions. That said, the `sim_venue` (given by the affine similarity) was yet to be normalised. To do so, we needed to know the range of the resulting `sim_venue` score, hence normalising was done after calculating the scores for each row across the entire dataset.

To implement pairwise comparisons between the `ACM` and `DBLP` dataset, we first tried a double-loop approach. Whilst functional, it resulted in a poor performing program. Thus, a vectorised approach on a cross-joined data frame was implemented:

```
def pairwise_comparison(df1: pd.DataFrame, df2: pd.DataFrame) -> pd.DataFrame:
    """
    Cross join the two dataframes and perform pairwise comparison through similarity
    scores. The venue score (calculated by the affine similarity) is normalised using the
    min-max formula. Similarity scores are applied in a vectorised method to increase
    performance for the O(n^2) algorithm. Finally, all the attribute scores are aggregated
    using custom weights. The resulting  score has a range from 0 to 1.
    :param df1: Dataframe left
```

```
    :param df2: Dataframe right
    :return: Cross-joined data with similarity score between records
    """
    # Reset ids
    df1 = df1.reset_index(names='id')
    df2 = df2.reset_index(names='id')

    # Initiate models.
    lev = sm.Levenshtein()
    jaro = sm.Jaro()
    aff = sm.Affine()

    # Cross join for pairwise comparison.
    df = pd.merge(df1, df2, how='cross')

    # Apply sim score calculations to rows (vectorised).
    df['sim_title'], df['sim_authors'], df['sim_venue'], df['sim_year'] =
        np.nan, np.nan, np.nan, np.nan
    df = df.apply(lambda r: record_sim(r, lev, jaro, aff), axis=1)

    # Apply min-max scaling (normalisation) to sim_venue.
    df['sim_venue'] = (df['sim_venue'] - df['sim_venue'].min()) /
        (df['sim_venue'].max() - df['sim_venue'].min())

    # Calculate similarity score.
    df['sim_score'] = 0.45 * df['sim_title'] + 0.45 * df['sim_authors'] + 0.05 *
        df['sim_venue'] + 0.05 * df['sim_year']
    return df
```

We first insert the indices of the dataframes back to an actual attribute called `id`. The reason for this is to retain the index data when we cross-join the two dataframes. This is done in the following code:

```
# Reset ids
df1 = df1.reset_index(names='id')
df2 = df2.reset_index(names='id')
```

We then make instances of the `py_stringmatching` models, as creating these per row would result in a longer runtime:

```
# Initiate models.
lev = sm.Levenshtein()
jaro = sm.Jaro()
aff = sm.Affine()
```

As mentioned, rather than using a double for-loop structure, we opted to use the `pandas.Dataframe.apply()` method to apply our record similarity calculation in a vectorised manner. The custom function described in section d.

assumes similarity score attributes are already in place, hence these need to be created as well before running the vectorised code. We initialise these attributes with NaN values. To facilitate the pairwise comparisons, we cross-join the two dataframes into one. These 3 steps are reflected in the following code:

```
# Cross join for pairwise comparison.
df = pd.merge(df1, df2, how='cross')
# Apply sim score calculations to rows (vectorised).
df['sim_title'], df['sim_authors'], df['sim_venue'], df['sim_year'] =
np.nan, np.nan, np.nan, np.nan
df = df.apply(lambda r: record_sim(r, lev, jaro, aff), axis=1)
```

Finally, we normalise the `sim_venue` data to a range of $[0, 1]$ (now knowing the full ranges) and compute the final similarity score:

```
# Apply min-max scaling (normalisation) to sim_venue.
df['sim_venue'] = (df['sim_venue'] - df['sim_venue'].min()) /
    (df['sim_venue'].max() - df['sim_venue'].min())

# Calculate similarity score.
df['sim_score'] = 0.45 * df['sim_title'] + 0.45 * df['sim_authors'] + 0.05 *
    df['sim_venue'] + 0.05 * df['sim_year']
```

Rather than returning a list of ids, we instead return a Dataframe object containing the cross-joined records and their respective similarity score. There were two main reasons for not returning a list in this method:

- Cross-joining the data implicitly meant working with Dataframe objects.
- Joining the similar records with the true mappings (as described in j.) is intuitive through the `pandas.merge()` method, avoiding the requirement for custom list matching.

j. Find the precision of the method compared to actual duplicate records.

We joined the records with *similarity* > 0.7 with the true mapping dataset. A new attribute `match` was added to the joined dataset, indicating whether the similarity score threshold correctly identified a duplicate record (1 = true, 0 is false). Due to the numeric property of this attribute, we could simply sum over the column to calculate the precision of the program. These steps are reflected in the following function:

```
def join_mapping(df_sims: pd.DataFrame, df_mapping: pd.DataFrame) -> pd.DataFrame:
    """
    Join the results of similar records to the true duplicate record data.
    A 'match' column is added to indicate whether a match was correct.
    :param df_sims: Dataframe containing similarity scores of cross-join
```

```
    :param df_mapping: Dataframe containing the true duplicate records
    :return: Dataframe with indicator of correct results
    """
    # Retrieve ids only.
    df_sims = df_sims[['id_x', 'id_y']].rename(columns={'id_x': 'idDBLP', 'id_y': 'idACM'})

    # Left join to indicate correct predictions.
    df_mapping['match'] = 1
    df_match = pd.merge(df_sims, df_mapping, how='left', on=['idDBLP', 'idACM'])
    df_match['match'] = df_match['match'].fillna(0)
    return df_match
```

The resulting data suggested that 2603 rows had a similarity score of > 0.7, 2187 of those records were correctly identifying true duplicate records, therefore giving a precision of 84%. It is interesting to note that the true mapping dataset (obtained from the `DBLP-ACM_perfectMapping.csv` file) had 2224 entries. This means that almost every duplicate record was identified by the similarity score threshold. The results also indicate that most of the inaccuracy is related to the amount of false positives generated by the similarity threshold. Increasing the threshold could therefore potentially increase the precision of the program.

k. Record the running time and propose a way to reduce it.

TODO replace the X with amount of records.

Initially, our code consisted of the functions to compute similarities found in the labs. Using the time package we measured that in the initial form the code took 6.7 seconds to execute a single comparison of a record in ACM data frame to DBLP data frame. We quickly noticed that running the entire comparison would require more than 4 hours. With such long running time, the chance of an error occurring and forcing us to repeat the lengthy process was quite high. We considered catching all of the errors and continuing with the process, however that is not a responsible practise.

Therefore, we decided to use likely more optimised functions found in the py_stringmatching pakage. In the more optimised version, the total run time of the detecting duplicate records algorithm was 33 minutes and 34 seconds. The time was found by the following code:

```
# Find similar records.
time_start = time.time()
df_sims = pairwise_comparison(df_dblp, df_acm)
duration = round(time.time() - time_start, 2)

# Runtime
print(f"Runtime of the pairwise similarity comparison is {duration} seconds.")
```

Still, the run time of the comparison remained quite high. Possible further solutions we considered included optimizing this method all included reducing

the number of comparisons executed. Specifically, by finding ways to find "candidate pairs". One way of doing so, is by introducing LHS method of creating similar buckets based on shingle signatures (implemented in Part 2).

We also considered creating blocks based on `year` or `venue` value and then comparing records that have identical values of the chosen attribute and vastly reducing the number of comparisons needed. In this case a small number of unique cases in both attributes is a benefit, as this method would not be beneficial in case of `title` or `authors`, where the number of different values is almost identical to the number of records. It is important to note, that in case of `venue` it would not be as simple as finding identical names in the two datasets, since some of the conference names are in the form of an acronym in the ACM dataset. Thus, either manual mapping or a similarity measure would be needed in the classification of blocks.

Part 2

1. Concatenate the values in each record into one single string.

This step was done in the preprocessing function. This function takes the datasets, and applies a rowwise function to take all the values in the row, and combine them into one string. The preprocessing function can be seen below.

```
def preprocess_df(df: pd.DataFrame) -> pd.DataFrame:
    """
    Preprocess the data by combining all columns for a row into one string,
    making these strings all lower case, and removing multiple spaces.
    """
    # join columns into one string
    df = df.apply(lambda row:
                  ' '.join(row.values.astype(str)),
                  axis=1)
    # Change all strings to lowercase
    df = df.str.lower()
    # Convert multiple spaces to single spaces
    df = df.str.replace(r'\s+', ' ', regex=True)
    return df
```

The code under the comment 'join columns into one string' is the code used to combine all column values in a row into one string. The function was applied on both the acm dataset, as well as the dblp dataset.

2. Change all alphabetical characters into lowercase.

The code for this assignment is also part of the preprocess_df function described above. The code under the comment 'Change all strings to lowercase' was used to ensure the singular string per row is lowercase in its entirety. As mentioned before, this code was applied to both the acm dataset and the dblp dataset.

3. Convert multiple spaces to one.

The code for this assignment is yet again part of the preprocess_df function described above. The code under the comment 'Convert multiple spaces to single spaces' was used to remove all multiple spaces. Using the regular expression r'\s+', all multiple spaces are removed, including tabs and new lines.

4. Combine the records from both tables into one big list.

To combine the records from both tables into one big list, the get_sentences function was created:

```
def get_sentences(df1: pd.DataFrame, df2: pd.DataFrame) -> list:
    """Take two dataframes and combine them into one sentence list"""
    sentences = df1.tolist() + df2.tolist()
    return sentences
```

This function takes two pandas data frames as input, and returns a list. This is done by casting both data frames to list using the toList() function, and then concatenating the resulting lists together using the plus operator. For this assignment, the preprocessed acm and dblp datasets were used as input for the function.

5. Use the functions in the tutorials from lab 5 to compute the shingles, the minhash signature and the similarity.

As mentioned in the assignment, the functions from lab 5 were used to implement the entire lsh algorithm, including the functions for creating the shingles, minhash and similarity. The functions used from the lab are

- The shingle function
- The build_vocab function
- The one_hot function
- The get_minhash_arr function
- The get_signature function
- The jaccard_similarity function
- The compute_signature_similarity function
- The entire LSH class including it's functions

To build the shingles, the shingles function was used to create shingles for each of the sentences in the sentences list. A k of 5 was used, meaning all shingles have a lenght of 5. With this shingles list, a vocabulary was build using the given build_vocab function. With both the shingles and vocab list the one-hot encoded vector matrix could be build.

Using the get_minhash_arr function, the minhash array was build. 800 hash-functions were used to ensure the number of candidate pairs created from the signatures would exceed 2224. With this minshash array, a list of signatures was made.

The assignment then asked to compute the similarity. We interpreted that as computing the similarity scores between each and every signature. The algorithm actually does not need these numbers, so it will heavily influence the performance. To compute the similarity scores between every signature, a similarity matrix was made using the function compute_similarity_matrix:

```
def compute_similarity_matrix(signatures: list) -> np.matrix:
    """
    Build a similarity matrix filled with the estimated jaccard similarity
    of two signatures.

    :param signatures: The list containing signatures that need to be
    cross-compared
    :return: The similarity matrix of all signatures
    """
    # Compute the similarity scores between all pairs of signatures
    # and store them in a matrix
    sim_scores = np.zeros((len(signatures), len(signatures)))
    for m in range(len(signatures)):
        for n in range(m + 1, len(signatures)):
            # If already visibly visited, skip
            if sim_scores[m, n] != 0:
                continue
            # If not yet visited, or the score is 0, compute
            sim_scores[m, n] = compute_signature_similarity(signatures[m],
                                                            signatures[n])
            sim_scores[n, m] = sim_scores[m, n]
    print(sim_scores)
```

First, and empty array is made that can hold the number on signatures on both axes. Then the scores are computed for each combination of signature. The matrix is also mirrored in the process, because the similarity score between signature m and signature n is the same as the similarity score between signature n and signature m. This allows for some speedup, because the function can skip computing similarity scores when the value is already computed. The result of this function is the similarity score matrix showing all possible similarity score combinations. This entire process takes about two minutes, thus influencing the running time of the method.

6. Extract the top 2224 candidates from the LSH algorithm, compare them to the actual mappings in the file DBLP-ACM_perfectMapping.csv and compute the precision of the method.

The LSH class from the practical was used, and lsh was applied on the signature list. 100 buckets were used when performing local-sensitive hashing. With the 100 buckets and 800 hashing functions, a total of 2397 candidate pairs were found. The top 2224 candidates were extracted, and put into a data frame to allow comparison to the perfect mapping data set. The perfect mapping dataset was also preprocessed by casting all values to strings, making them lowercase and removing multiple spaces. Both datasets were then cast to sets, and the precision was calculated by dividing the number of elements in the intersection of the sets by 2224, the length of both sets. This resulted in a precision of 50%. It should be noted that when tweaking the number of hash functions used and the number of buckets, the precision of this method could be improved without influencing the running time of the program too much, as lsh runs very fast.

7. Record the running time of the method.

The timer was started after preprocessing the datasets, as this was also done in task 2.1. The timer was thus started before the shingles were computed. The timer was stopped after the precision was calculated. With 100 buckets and 800 hashing functions, the program took roughly 40 seconds to run, excluding the time it took to compute the similarity matrix. When including the building of the similarity matrix, the running time is roughly 2 minutes and 40 seconds.

8. Compare the precision and the running time in Parts 1 and 2.

The method used in task 2.1 had a precision of 84% and a running time of 33 minutes and 34 seconds. The method using local-sensitive hashing had a lower precision of 50% with a running time of 2 minutes and 40 seconds when computing the similarity matrix, and just 40 seconds when only applying the algorithm. The precision of the method used in task 2.1 is a lot higher, but the speed-up that lsh provides, makes local-sensitive hashing much more user friendly. The precision when using local-sensitive hashing could potentially be improved further when tweaking the parameters, but the time this would require would still be far less then the time it would take to perform pairwise comparison. Overall, when wanting the highest precision possible without having to worry about time, pairwise comparison would be the best method. But when time becomes a parameter in this equation, local-sensitive hashing seems like the much speedier option, with a decent precision to match.

Task 3: Data preparation

This section explains the steps taken to prepare the data and perform correlation analysis as per the task requirements. The dataset used is the Pima Indians

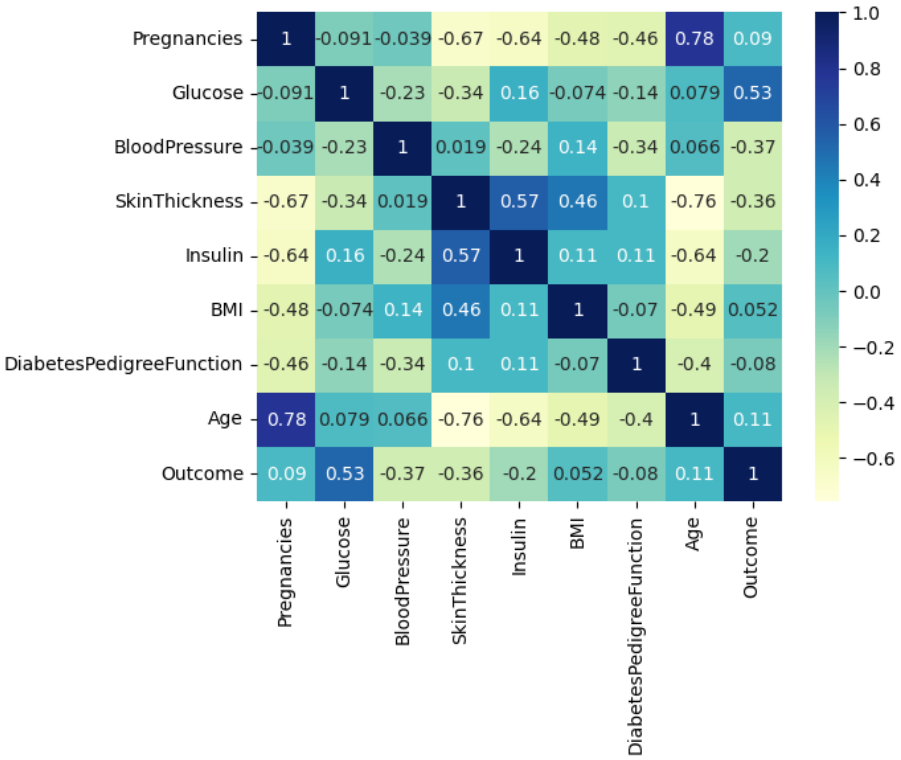

Figure 1: Correlation matrix

Diabetes Dataset, and the steps include handling missing values, filling them using mean values, and observing changes in correlation between features.

3.1 Compute correlation of columns other than outcome.

We first compute the correlation matrix between all columns, excluding the 'Outcome' column. This serves as the baseline for the analysis, and it will later be compared with the final correlation matrix after handling missing values.

The matrix was generated with following code (result in figure 1):

```
diabetes_data.drop(columns=['Outcome'])
correlation_before = diabetes_data.corr()
dataplot = sb.heatmap(correlation_before.corr(numeric_only=True),
    cmap="YlGnBu", annot=True)
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	<NA>	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1
5	5	116	74	<NA>	0	25.6	0.201	30	0
6	3	78	50	32	88	31.0	0.248	26	1
7	10	115	<NA>	<NA>	0	35.3	0.134	29	0
8	2	197	70	45	543	30.5	0.158	53	1
9	8	125	96	<NA>	0	<NA>	0.232	54	1
10	4	110	92	<NA>	0	37.6	0.191	30	0

Figure 2: Dataset with null values

3.2 Remove disguised missing values.

Disguised missing values in the 'BloodPressure', 'SkinThickness', and 'BMI' columns were represented by the value '0'. These '0' values were replaced by 'NaN' to correctly represent missing data (result in figure 2:

```
columns_to_replace = ['BloodPressure', 'SkinThickness', 'BMI']
diabetes_data[columns_to_replace] = diabetes_data[columns_to_replace].replace(0, pd.NA)
```

3.3 Impute with mean values.

The missing values ('NaN') in 'BloodPressure', 'SkinThickness', and 'BMI' were filled using the mean values of records grouped by the 'Outcome' column. This ensures that missing values are filled based on the class label (whether the patient has diabetes or not).

```
for column in columns_to_replace:
    diabetes_data[column] = diabetes_data.groupby('Outcome')[column].transform(lambda x:
        x.fillna(x.mean()))
```

Figure 3 shows the imputed dataset.

3.4 Compute correlation matrix of imputed dataset.

After handling the missing values, the correlation matrix was recomputed to observe any changes in relationships between the variables. The resulting matrix can be seen in figure 4.

```
correlation_after = diabetes_data.corr()
dataplot = sb.heatmap(correlation_after.corr(numeric_only=True), cmap="YlGnBu", annot=True)
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72.0	35.0	0	33.6	0.627	50	1
1	1	85	66.0	29.0	0	26.6	0.351	31	0
2	8	183	64.0	33.0	0	23.3	0.672	32	1
3	1	89	66.0	23.0	94	28.1	0.167	21	0
4	0	137	40.0	35.0	168	43.1	2.288	33	1

Figure 3: Imputed dataset

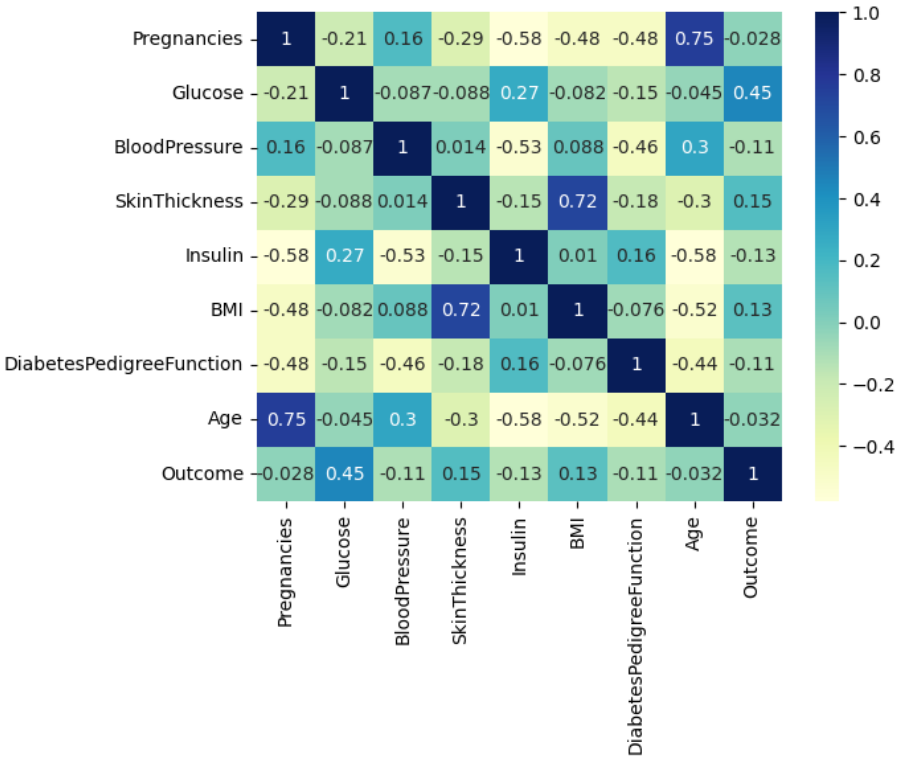

Figure 4: Correlation matrix of imputed dataset

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
Pregnancies	0.000000	0.000000	0.067653	0.175844	0.000000	0.006444	0.000000	0.000000	0.000000
Glucose	0.000000	0.000000	0.069828	0.163615	0.000000	-0.001192	0.000000	0.000000	0.000000
BloodPressure	0.067653	0.069828	0.000000	-0.003918	-0.137039	0.004713	-0.043529	0.084911	0.110019
SkinThickness	0.175844	0.163615	-0.003918	0.000000	-0.332766	0.172869	-0.081501	0.249886	0.233342
Insulin	0.000000	0.000000	-0.137039	-0.332766	0.000000	-0.012314	0.000000	0.000000	0.000000

Figure 5: Change in matrices

3.5 Correlation matrix comparison.

The initial and final correlation matrices were compared to see how the relationships between variables changed after handling the missing data.

```
correlation_change = correlation_after - correlation_before

correlation_before, correlation_after, correlation_change
correlation_change.head()
```

The above code revealed the changes between the two matrices, as can be seen in figure 5.

Significant changes were observed in 'BloodPressure', 'SkinThickness', and 'BMI', where missing values were handled.

The correlation matrices before and after handling missing values show some noticeable changes, particularly in how key features relate to each other and to the Outcome column.

BloodPressure

Before handling missing values, the correlation of BloodPressure with several other features like Pregnancies, Glucose, and Age was relatively moderate. However, after imputing missing values, these correlations increased slightly. This suggests that after replacing the missing BloodPressure values (which were previously marked as 0), there is a stronger relationship between BloodPressure and features such as Pregnancies and Glucose.

One of the most significant changes is the correlation between BloodPressure and Insulin, which became more negative. This could indicate that after handling the missing values, Insulin and BloodPressure became more distinct in terms of their relationship to each other.

Perhaps the most interesting change is the correlation between BloodPressure and Outcome. Initially, the correlation was fairly low (around 0.065), but after handling the missing values, it jumped to 0.175. This suggests that BloodPressure now has a stronger association with whether or not a person has diabetes. This change could be due to the fact that missing BloodPressure values (which were likely replaced by the mean values) may have altered the distribution of the data, leading to a more accurate reflection of the relationship between BloodPressure and diabetes outcome.

SkinThickness

The SkinThickness variable also shows significant changes after handling missing values. Before the imputation, its correlation with key features such as Pregnancies, Glucose, BMI, and Outcome was lower. However, after filling the missing values, the correlations increased across the board. Specifically, the correlation between SkinThickness and Outcome rose substantially, from 0.07 to around 0.31. This suggests that after imputing the missing SkinThickness values, there is a much stronger relationship between SkinThickness and whether or not a person has diabetes.

This change likely reflects the fact that after missing values were replaced, the SkinThickness data became more representative of the entire dataset, allowing the feature to have a more accurate correlation with other variables and with the diabetes outcome.

SkinThickness

When it comes to BMI, the changes in correlation are less dramatic than those seen in BloodPressure or SkinThickness. However, there are still some notable changes. The correlation between BMI and SkinThickness, for example, increased from 0.392 to 0.565. This indicates that these two features are more strongly related to each other after handling missing values. Additionally, the correlation between BMI and Outcome increased slightly, from 0.292 to 0.315, suggesting a modest improvement in how well BMI predicts the likelihood of diabetes after missing values were handled.

Conclusion

In summary, the correlation changes highlight the fact that handling missing values can have a substantial impact on the relationships between variables. Most notably, the features BloodPressure and SkinThickness became more strongly correlated with the diabetes outcome (Outcome) after missing values were handled. These features, which previously had 0 values that acted as disguised missing data, now show a clearer relationship with diabetes after the imputation process.

BMI also showed some changes, although the shifts were less dramatic, which could indicate that it was less affected by missing data to begin with. The overall effect of handling missing values is an improved understanding of how the features relate to each other, and in particular, how they predict the diabetes outcome.