DSA8002: Database and Programming Fundamentals
Coursework Assignment Report

Emir Akgiun
Student Number - 40367624

This assignment report contains a full walk-through of all the work process that took place in order to complete the given tasks, supplemented by explanations, code snippets and auxiliary graphics.
Other files that comprise the complete coursework submission and that may be referenced in this report can be found in the submitted ZIP archive. A more thorough and itemized description of the submission contents and dependencies could be obtained by consulting with the *README.txt* file, found in the exact same archive.

Table of Contents

I Task 1

In the first task the key to developing a proper solution was a thoughtful and mindful utilization of *pandas* Data Frame objects and proper methods, allowing us to prepare, mutate and filter through the given sets of data. Overall, the entire output pertaining to the task is contained within the *Task1.py* source code file.

I.i Section A - Data Import

As a matter of fact, the first subsection of the task required mostly basic methods found in the *pandas* package. First and foremost, it is detrimental that we take a deeper look at the data we are provided with, so we can develop familiarity with it, make observations and potentially gain preliminary insight. Initially, we are given 3 files with comma-separated values:

- *movies.csv* - key file containing all the data giving a basic description of movies in our database, including their *movieId*,*title*,*genres*, separated by the "pipe" symbol and clearly having some genres written with lowercase first letter.
- *ratings.csv* - a file containing ratings, which the users left to each movie: *userId*,*movieId*,*rating*,*timestamp*.
- *tags.csv* - a file containing all the tags users assigned to each movie: *userId*,*movieId*,*tag*,*timestamp*.

To begin with, the files should be uploaded into data frames, which is easily done with the help of *pandas* methods. It is also quite important to ensure we have some idea of what we have uploaded. For that purpose we could print the shapes of the new data frames, thus obtaining the number of data points and columns, as well as making sure they have been uploaded correctly.

```
        """ Section A - Import"""
        moviesDf = pd.read_csv("movies.csv")
        ratingsDf = pd.read_csv("ratings.csv")
        tagsDf = pd.read_csv("tags.csv")
        print(f"{tagsDf.shape}")
        print(f"{ratingsDf.shape}")
```

Listing 1: Task 1(A) - file importing

Next up, in *Listings 2 and 3*, the pandas methods *df.dropduplicates()* and *df.dropna()* can be used to get rid of the duplicates and Na values in all data frames. While dropping Na values, it is possible to specify a subset of values, which, when repeated, result in duplicate rows being eliminated. In our case, user can not leave two reviews for one movie, and one movieId can not have multiple titles, so in *Listing 2* we use sub-setting for the movies and ratings data frames.

```
        # Next up, using the dataframes created it is necessary to
    perform some  data cleaning
        # Dropping duplicates
        moviesDf = moviesDf.drop_duplicates(subset=["movieId", "
    title"])
```

```
        ratingsDf = ratingsDf.drop_duplicates(subset=["movieId", "
    userId"])
        tagsDf = tagsDf.drop_duplicates()
```

Listing 2: Task 1(A) - duplicate removal

```
        moviesDf = moviesDf.dropna()
        ratingsDf = ratingsDf.dropna()
        tagsDf = tagsDf.dropna()
```

Listing 3: Task 1(A) - Na values removal

It may be important to mention, that in the source code itself some small tests were also added for ensuring that the cleaning was successful and there were no undesirable data points anymore. The final preparatory step is to eliminate all data points in ratings and tags data frames, whose movieId does not belong to any movieId in the original movie data frame. Thus, only the relevant reviews and tags are maintained. For that purpose the pandas *.loc()* method,which will allow for selection of specific rows, as well as *.isin()*, which compares the set of values to the value of an object.

```
# Lastly, filtering out entries in Ratings, Tags  that  do NOT
    pertain to any entries in Movies
        allMovieId = moviesDf["movieId"]
        ratingsDf = ratingsDf.loc[ratingsDf["movieId"].isin(
    allMovieId)]
        tagsDf = tagsDf.loc[tagsDf["movieId"].isin(allMovieId)]
```

Listing 4: Task 1(A) - irrelevant reviews and tags removal

I.ii Section B - Similar Movie Retrieval

Moving on to the next section, we are tasked with finding similar movies to the one selected by the user. As for now, it is necessary to retrieve all movies that have similar rating and at least one common genre. Using a *while* loop the user is prompted to enter a desired movieId, only proceeding if the movieId exists. After that, I chose to save all the parameters relating to the chosen movie, such as its genres, title, ratings and tags.

```
        allMovieId = allMovieId.tolist()
        selectedId = -1
        while (selectedId not in allMovieId) == True:
                if test == False:
                        print('Enter the Movie Id: ')
                        selectedId = int(input('> '))
                else:
                        print('Testing mode. Input predefined. ')
                        selectedId = int(testInput)
        # Saving the values of columns pertaining to the chosen
    movie
        selectedMovie = moviesDf['title'].loc[moviesDf['movieId']
    == selectedId].values.tolist()
```

```
        selectedGenre = moviesDf['genres'].loc[moviesDf['movieId']
    == selectedId].values
        selectedRatings = moviesDf['rating'].loc[moviesDf['movieId'
    ] == selectedId].values
        selectedTags = moviesDf['tag'].loc[moviesDf['movieId'] ==
    selectedId].values
```

Listing 5: Task 1(B) - User input prompting and saving parameters of the selected movie

When working with any conventional data objects it is necessary to follow certain formal requirements. So, in order for the merged data set to align with the First Normal Form, it is necessary to use *pd.explode()* on that column and normalize the data set. After that everything is ready for retrieving from the data frame using comparison operators and the *.isin()* method. The entire retrieval script can be see in the *Listing 6*.

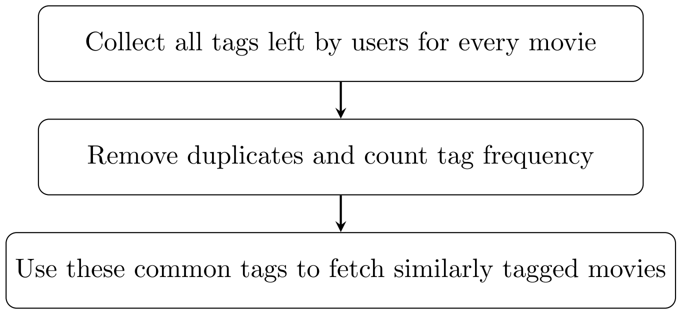
```
selectedGenre = list(selectedGenre)[0]
        selectedRatings = list(selectedRatings)[0]
        mergedDf = mergedDf.explode('genres')
        mergedDf = mergedDf.drop_duplicates(subset=["genres", "
    rating", "tag"])
        # Fetching movies with matching ratings and genres,
    dropping all the duplicate rows and  unnecessary columns
        similarMovies = mergedDf[mergedDf["movieId"] != selectedId]
        similarMovies = similarMovies[similarMovies['genres'].isin(
    selectedGenre)&similarMovies['rating'].isin(selectedRatings)]
        similarMoviesFinal = similarMovies.drop_duplicates(subset =
     ["movieId", "title"]).drop(labels = ["rating", "genres", "tag"
    ],axis = 1)
        print(similarMoviesFinal)
```

Listing 6: Task 1(B) - Exploding genres column to conform to 1NF and retrieving a dataframe of similar movies

As a result, we obtain a data frame with movies, which contain at least one common rating and genre and, usually, that is a pretty large number of similar movies. Luckily, the next task allows us to tweak the retrieval algorithm, so the similarities found narrow down the resulting data frame even further.

I.iii Section C - Enhanced Similar Movie Retrieval

In the third and final subsection of Task 1 it is necessary to enhance (i.e. make narrower) the retrieval algorithm from subsection B. Frankly, there are many ways in which tag-based movie retrieval could be implemented.
The algorithm implemented by myself for subsection C can be schematically summarized as follows:


Collect all tags left by users for every movie
Remove duplicates and count tag frequency
Use these common tags to fetch similarly tagged movies

The few lines at the beginning of subsection C of Task1.py, actually, do the same filtering as in subsection B and create a proper list of tags from the selected movie. After that we should calculate the frequency of every tag appearing.It could be done by using the pandas method *.size()* while grouping by movieId and tag.

```
    # Creating a dataframe containing the number of instances of each tag being  used on every movie
    similarMoviesFreq = similarMoviesEnhanced.groupby(['movieId','tag']).size()
    #If for some reason no dataframe is created, it is necessary to change that object's type
    if not isinstance(similarMoviesFreq, pd.DataFrame):
            similarMoviesFreq = similarMoviesFreq.to_frame()
```

Listing 7: Task 1() - tag frequency calculation

Then code below mutates the frequencies data frame so it can be joined in with the movies data frame later on.

```
    # Mutating the dataframes in an appropriate way
    similarMoviesFreq = similarMoviesFreq.reset_index()
    similarMoviesFreq = similarMoviesFreq.rename(columns={0: 'tag frequency'})
    similarMoviesEnhanced = similarMoviesEnhanced.drop_duplicates(subset=["tag"]).drop(labels = ["rating", "genres"],axis = 1)
    similarMoviesEnhanced =  similarMoviesEnhanced.reset_index()
```

Listing 8: Task 1() - frequency data frame preparation

Finally, we can find other movies that have similar tags to the ones we have in the similarMoviesFreq data frame. Movies are also sorted by their similar tag frequency. However, if no movie fitting all the criteria can be fetched (i.e. returning an empty data frame), the conditional statement prints out a relevant prompt and sets the value of similarMoviesEnhancedFinal* to None.
this variable is returned by function, however, previously in Task 2 a table of similar movies was still printed out and could be used to identify more or less similar movies anyway.
In case the search was a success, the function Task1() returns a filled data frame and prints it, dropping all columns except movieId and movie titles.

```
    # Checking whether any similar tag frequencies were added and append them to the dataframe/sort the frequencies
    if not similarMoviesFreq.empty:
            similarMoviesEnhanced = similarMoviesEnhanced.join(similarMoviesFreq['tag frequency'])
            similarMoviesEnhanced= similarMoviesEnhanced.sort_values('tag frequency', ascending=False)
    # Final output of the function, the tag-coincidence rate and a skimmed best-matching movies dataframe
    if similarMoviesEnhanced.empty:
            print(f"No movies of similar genre and rating share same tags as the movie you chose :( ")
            similarMoviesEnhancedFinal  = None
    else:
            print(f"Here are the matching movies of the same genre(s) and similar ratings, sorted  by common tag frequency: ")
            print(similarMoviesEnhanced)
            print(f"Finally, just the best matching movies: ")
            similarMoviesEnhancedFinal = similarMoviesEnhanced.drop_duplicates(subset=["movieId", "title"]).drop(labels=["tag", "tag frequency","index"], axis=1)
            print(similarMoviesEnhancedFinal)
    return similarMoviesEnhancedFinal
```

Listing 9: Task 1() - finding similar tags

That snippet finalizes the entire task, returning a variable *similarMoviesEnhancedFinal* in case tag-based similarities are encountered, which is a data frame with movie titles and movieIds. Anyhow, below the Task1() function body and its call there is a chunk of code that performs a functional test of the task function. More on that test could be found in the *Task4* section.

II Task 2

In the following task the main goal is to create a series of SQL statements so as when they are executed, they create tables equal to data frames used previously. The only exception is that a separate table for genres should be created as well, as storing it within the movies table is inappropriate and violates 1NF. Also, the string values in data frames should be wrapped in quotation marks so they are passed properly into the mask and do not create any SQL injections e.g. SQL reserved keyword ACTION could be ambiguous with "Action" as a genre. For the purposes of generation of a bunch of working SQL script files, Python should come in handy.

```
        def createSQL(dataFrame, tableName):
            # Obtaining the list of columns and data types  in the dataframe
            columns = []
            for col, dtype in zip(dataFrame.columns, dataFrame.dtypes):
                    if str(dtype) == 'object':
```

```
                                dtype = 'VARCHAR(255)'
                        elif 'int' in str(dtype):
                                dtype = 'INT'
                        elif 'float' in str(dtype):
                                dtype = 'FLOAT'
                        elif 'datetime' in str(dtype):
                                dtype = 'DATETIME'
                        # Creating a list of column names and the data types of values in it
                        columns.append(f"{col} {dtype}")
                # Creating the table
                columnStr = ", ".join(columns)
                createTable = f"CREATE TABLE {tableName} ({columnStr});"
                #  Finally, we can write the created strings into a .sql file
                with open(f'{tableName}.sql', 'wb') as f:
                        # Write the CREATE TABLE statement
                        f.write(createTable.encode('utf-8'))
                        # The string is binary to facilitate proper writing to the document
                        f.write(b'\n')
                        # Building and writing the INSERT INTO statements, looping  through the dataframe
                        for _, row in dataFrame.iterrows():
                                insert = f"INSERT INTO {tableName}({', '.join(dataFrame.columns)}) VALUES ({', '.join([str(x) for x in row])});"
                                f.write(insert.encode('utf-8'))
                                f.write(b'\n')
```

Listing 10: Task 2 - Python script for generating SQL statements

The code above first checks data types in columns in the data frame that was passed as argument to the function *createSQL* and creates a list consisting of column names and data type, but written in a SQL style. Then it creates and writes the CREATE TABLE statement with all the previously created values of column names and data types. Lastly, iterating on all the rows in the argument data frame, the function writes INSERT INTO statements according to the mask. Below is the example of a few lines from the newly generated movies.sql script.

```
       CREATE TABLE movies (movieId INT, title VARCHAR(255));
INSERT INTO movies(movieId, title) VALUES (1, "Toy Story (1995)");
INSERT INTO movies(movieId, title) VALUES (2, "Jumanji (1995)");
INSERT INTO movies(movieId, title) VALUES (3, "Grumpier Old Men (1995)");
INSERT INTO movies(movieId, title) VALUES (4, "Waiting to Exhale (1995)");
INSERT INTO movies(movieId, title) VALUES (5, "Father of the Bride Part II (1995)");
```

Listing 11: Task 2 - Generated SQL statements example: movies.sql

III Task 3

To begin with, Task 3 is quite similar to Task 1 in its gist, with the only large difference being that in this case it is necessary to use SQL queries to retrieve similar movies. *Pandasql* package is used for that purpose, so as to allow for operating over existing data frames. The general algorithm of cleaning and preparing data in Task 3 is, as can be observed in the source code files, almost identical to the same

III.i Section A - Similar Movie Retrieval with SQL

Importantly, it could be noticed that the SQL queries are formatted strings in my source code. Making the string formatted allows us to insert variables/functions from Python into SQL statements. In the query below, we assign

```sql
SELECT DISTINCT m.movieId, m.title
FROM moviesDf m
JOIN ratingsDf r ON m.movieId = r.movieId
JOIN genresDf g ON m.movieId =  g.movieId
WHERE r.rating IN (SELECT DISTINCT rating from ratingsDf r WHERE r.
    movieId = {selectedId})
AND g.genres IN (SELECT DISTINCT genres  FROM genresDf g WHERE g.
    movieId = {selectedId})
AND m.movieId != {selectedId};
```

Listing 12: Task 3 - SQL over dataframes to fetch similar movies

III.ii Section B - Enhanced Similar Movie Retrieval with SQL

In order to add tagsDf into the mix it is necessary to add one more JOIN statement and add 1 more AND to the WHERE clause. In result, the SQL query below outputs a more concise return, while getting rid of movies with no similar tags to the selection.

```sql
SELECT DISTINCT m.movieId, m.title
FROM moviesDf m
JOIN ratingsDf r ON m.movieId = r.movieId
JOIN genresDf g ON m.movieId =  g.movieId
JOIN tagsDf t ON m.movieId = t.movieId
WHERE r.rating IN (SELECT DISTINCT rating from ratingsDf r WHERE r.
    movieId = {selectedId})
AND g.genres IN (SELECT DISTINCT genres  FROM genresDf g WHERE g.
    movieId = {selectedId})
AND t.tag IN (SELECT DISTINCT tag  FROM tagsDf t WHERE t.movieId =
    {selectedId})
AND m.movieId != {selectedId};
```

Listing 13: Task 3 - SQL over dataframes to fetch similar movies with tags

That returns a data frame with all data points satisfying the conditions and ,thus, sums up Task 3. There is also a test chunk of code (which is covered in Task 4 section) below main task code in the same file.

IV Task 4

As was previously mentioned, I made a decision to implement functional tests at the end of every task, which I personally found to be a more logical approach as long as the tasks are evaluated separately and sequentially. However, in the source code file *Task4.py* all the code snippets are combined for the purpose of demonstration of the tests implemented for the fourth. In tasks 1 and 3, the functions containing all operations

IV.i Test Case 1

As for the first task, I developed a test case where the theoretical user tries to find a film similar to the "The Right Stuff" by inputting a movieId. In this case, Task1() function is passed arguments: (test = True, inputValue = 1231). That disables the user input solicitation and the function uses the inputValue as the movieId. The variable *expectedOutput* is also filled manually. The test checks whether the return of the function Task1() is identical to the expected output. The test should return successfuul if the output of the funtion and the expected return are equal.

```
""" Test for Task 1 """
"""Test Case 1: A user tries to find a similar movie to the one
    with ID 1231 "The Right Stuff". The output should contain a
    movie titled "Apollo 13", which has an ID of 150"""
from Task1 import Task1
testValue = 1231
expectedOutput = pd.DataFrame({"movieId": 150, "title": "Apollo 13
    (1995)"}, index=[0])
testOutput =  Task1(test = True, testInput = testValue)
if testOutput.equals(expectedOutput):
        print("Test 1 passed successfully.")
else:
        print("Test 1 failed.")
```

Listing 14: Code for Test Case 1

IV.ii Test Case 2

As for the second task, I developed a test case to check whether the functionality of the script generating the SQL statements is working properly and builds correct tables. In order to actually build a database, a module *sqlite3* found in standard python library is used. After creating a local connection to a database file, it is possible to execute SQL in it. For the sake of the test, I only run SQL files containing CREATE TABLE and INSERT INTO statements in movies.sql and genres.sql, so it is possible to execute a query in the question. When this test is run in the Task2.py script, the test should be successful.

```
""" Test for Task 2 """
"""Test Case 2: A database developer wants to fetch movie title and
     genre with moviedId set to 2  """
```

```
#It is necessary to  set up a local database in order to create SQL
     tables with our files
conn = sqlite3.connect("testdatabase.db")
cursor = conn.cursor()
#Executing our generated SQL files. !!! IF DATABASE ALREADY EXISTS,
     COMMENT OUT THIS SECTION
# with open('movies.sql', 'r') as f:
#         sql1 = f.read()
#         cursor.executescript(sql1)
# with open('genres.sql', 'r') as f:
#         sql2 = f.read()
#         cursor.executescript(sql2)
# Commiting the changes to the database
conn.commit()
#END OF DATABASE CREATION!!!

# Selecting all data needed on movieId 2 into a dataframe"
testMoviesDf = pd.read_sql("SELECT title, genres FROM movies,
    genres WHERE movies.movieId == 2 AND genres.movieId == 2", conn
    )
print(testMoviesDf.head())
expectedOutput = pd.DataFrame({"title": ['Jumanji (1995)','Jumanji
    (1995)','Jumanji (1995)'], "genres": ["adventure", "children",
    "fantasy"]}, index=[0,1,2])
# Comparing to expected behaviour
if testMoviesDf.equals(expectedOutput):
        print("Test 2 passed successfully.")
else:
        print("Test 2 failed.")
```

Listing 15: Code for Test Case 2 including commented out database creation block

IV.iii Test Case 3

As for the third task, I developed a test case that is similar to the one in Task 1.

```
""" Test for Task 3 """
"""Test Case 3: A user aims to find a similar movie to the one with
     ID 2. """
expectedOutput = pd.DataFrame({"movieId": [4993, 7153, 46972,
    59501, 80834, 106489], "title": ["Lord of the Rings: The
    Fellowship of the Ring, The (2001)","Lord of the Rings: The
    Return of the King, The (2003)","Night at the Museum (2006)","
    Chronicles of Narnia: Prince Caspian, The (2008)", "Sintel
    (2010)", "Hobbit: The Desolation of Smaug, The (2013)"]}, index
    =[0,1,2,3,4,5])
testOutput = Task3(test = True, testInput= 2)
if testOutput.equals(expectedOutput):
        print("Test 3 passed successfully.")
else:
        print("Test 3 failed.")
```

Listing 16: Code for Test Case 3

That concludes this report on the assignment. All the tests should pass successfully as long as they run at the end of their respective task's source code. Overall, we successfully managed to prepare and wrangle the provided data, perform necessary retrieval and filtering tasks as well as to build SQL script files that create SQL tables from data frames and to test the functionally of the source code.