DSA8002 Coursework Assignment Report

Introduction

In this project report, we will describe how we solved these tasks in detail with the three given CSV files. These three CSV files store information about movies, which are movies.csv, ratings.csv and tags.csv.

There are three fields in the movies.csv file containing the *movieId*, *title* and *genres*. This file gives the basic information about each movie, like the unique id, the title and the genres. The id is unique, which means one id corresponds to one movie, you will never find a few movies that share one id, on the contrary, it is impossible to get a movie with multiple ids. The string in the genres field is a list of genres separated by the pipe character ("|") related to the movies.

Within the ratings.csv file, we would know how users rated movies. Each user has a unique id like movies called *userId*, like a movie. The *movieId* in this case is a foreign key to movies.csv. The *rating* is a number given by the user to the movie and the *timestamp* is the time to leave a rating. The bigger the rating number, the better the movie users thought.

The tags.csv is similar to the ratings.csv and contains information about users leaving tags. *userId* and *movieId* are exactly the same as ratings.csv. *tag* is a string left by the user at the time the *timestamp* was recorded.

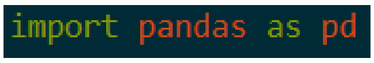
Based on these three CSV files, we will implement Python or SQL code for each task of this assignment.

Task 1

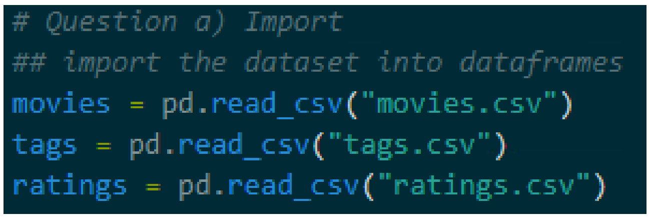
Question a)

We broke down this question into four steps. To start with, we import all the files into a dataframe. Then we remove rows containing duplicates. Next, we find all the rows containing blanks and remove them as well. Finally, we will try to find the movies that are in ratings.csv or tags.csv but not showing up in the movies.csv and delete them.

We import three CSV files by using "pd.read_csv()" with importing "pandas" as "pd". Since all the files are in the same folder, we just need to type the name of files instead of the whole path.

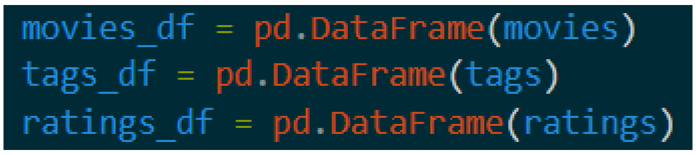


```
import pandas as pd
```



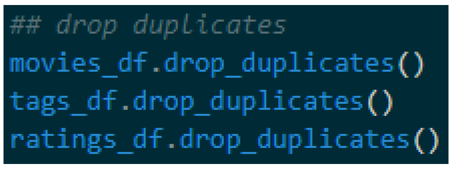
```
# Question a) Import
## import the dataset into dataframes
movies = pd.read_csv("movies.csv")
tags = pd.read_csv("tags.csv")
ratings = pd.read_csv("ratings.csv")
```

And then save them as dataframes with "pd.DataFrame()".



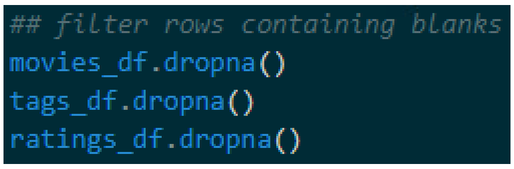
```
movies_df = pd.DataFrame(movies)
tags_df = pd.DataFrame(tags)
ratings_df = pd.DataFrame(ratings)
```

After this, we drop all the duplicate rows in these dataframes with "df.drop_duplicates()" since duplicated data could be a big problem when you try to aggregate data later. "df" here means the dataframe we got from the last step.



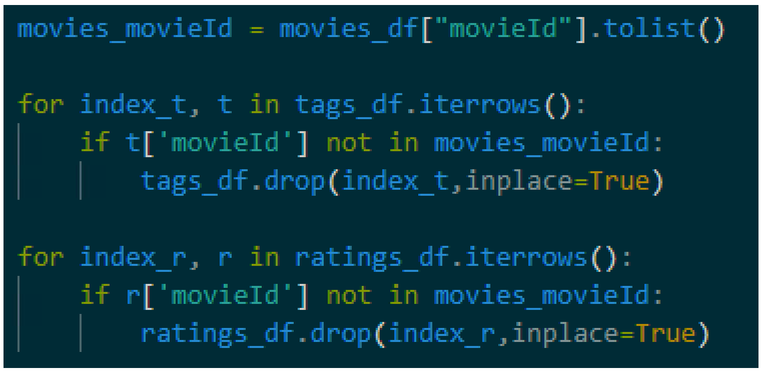
```
## drop duplicates
movies_df.drop_duplicates()
tags_df.drop_duplicates()
ratings_df.drop_duplicates()
```

Later, filter rows containing blanks.



```
## filter rows containing blanks
movies_df.dropna()
tags_df.dropna()
ratings_df.dropna()
```

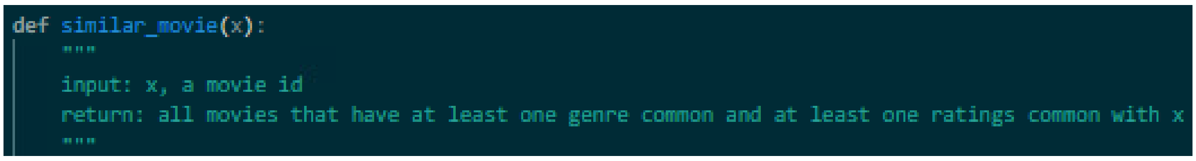
The most significant part of this sub-question is to ensure that all movies in ratings.csv and tags.csv files exist in the movies. We save all of the movieIds as a list and then compare them to the movieIds of other files. We use the iteration function "df.iterrows()" to help, when the movieid of that row is not in the list, which means their movieId does not exist in the movies file, we drop that row.



```python
movies_movieId = movies_df["movieId"].tolist()

for index_t, t in tags_df.iterrows():
    if t['movieId'] not in movies_movieId:
        tags_df.drop(index_t,inplace=True)

for index_r, r in ratings_df.iterrows():
    if r['movieId'] not in movies_movieId:
        ratings_df.drop(index_r,inplace=True)
```

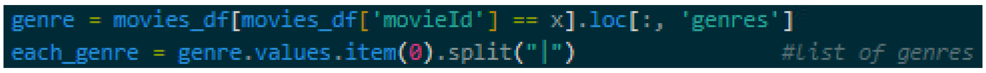
Question b)

In this question, we will look for similar movies to the given movie ID that share at least one genre and one rating. We create a function called "similar_movie", where the input parameter is the movieId we search similar movies for, and the returning result is a list of movies that match the condition.



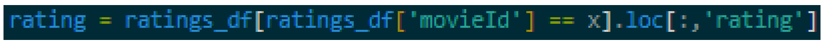
```python
def similar_movie(x):
    """
    input: x, a movie id
    return: all movies that have at least one genre common and at least one ratings common with x
    """
```

To start, we must identify the genres and ratings we are seeking. To split the genre string into its component parts, we use the "split()" function after retrieving the genre string based on the movieId.



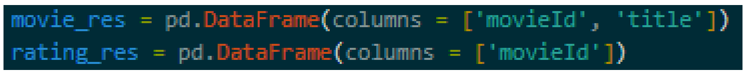
```python
genre = movies_df[movies_df['movieId'] == x].loc[:, 'genres']
each_genre = genre.values.item(0).split("|")            #List of genres
```

In a similar way, we obtained a series of ratings with the movieId. However, since it is not a string that contains multiple ratings, we do not need to split it at this point; we can simply save it in the "rating" variable.



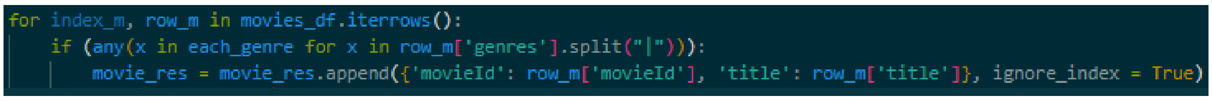
```python
rating = ratings_df[ratings_df['movieId'] == x].loc[:,'rating']
```

We create two new dataframes to save the movies after they have been filtered with "pd.DataFrame()". Movies that have at least one genre in common will be saved in the variable "movie_res", and those that have at least one identical rating will be saved in "rating_res". By taking the common part of these two dataframes, the answer will be obtained.



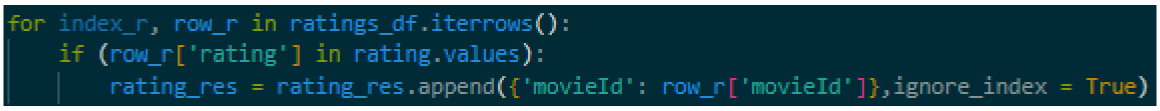
```python
movie_res = pd.DataFrame(columns = ['movieId', 'title'])
rating_res = pd.DataFrame(columns = ['movieId'])
```

By iterating through "movies_df", splitting the genres of each row, and comparing the split list with our target "each_genre" list, if any of it appears in the "each_genre" list, we will add the movieId and title of this row to the "movie_res".



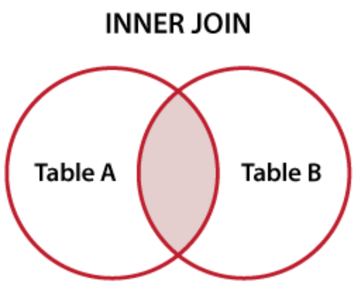
```python
for index_m, row_m in movies_df.iterrows():
    if (any(x in each_genre for x in row_m['genres'].split("|"))):
        movie_res = movie_res.append({'movieId': row_m['movieId'], 'title': row_m['title']}, ignore_index = True)
```

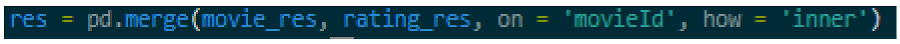
We apply the same strategy to the ratings as well. We chose to save movieId only here because it is unique and can lead to a specific movie, and adding more columns will result in duplicate columns after joining. Following that, we will have two dataframes containing a list of movies.



```
for index_r, row_r in ratings_df.iterrows():
    if (row_r['rating'] in rating.values):
        rating_res = rating_res.append({'movieId': row_r['movieId']},ignore_index = True)
```

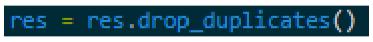
These two dataframes were inner joined using the same "movieId" column to produce the desired outcomes.





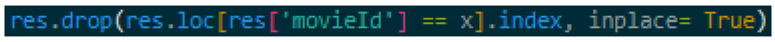
```
res = pd.merge(movie_res, rating_res, on = 'movieId', how = 'inner')
```

Duplicated rows are removed once again to guarantee there are not two same rows in the "res".



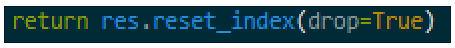
```
res = res.drop_duplicates()
```

The second-last step in completing this function is to drop the row whose movieId is input since we would prefer not to include the searched id itself in the result.



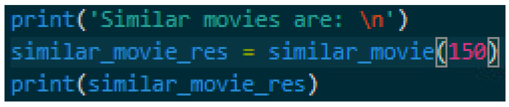
```
res.drop(res.loc[res['movieId'] == x].index, inplace= True)
```

Last but not least, just before returning the result, reset the index and let it be gradually added.



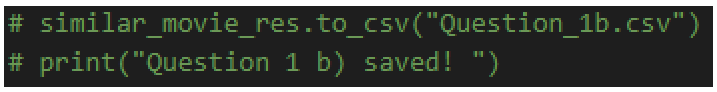
```
return res.reset_index(drop=True)
```

Then we call the function and type in the movieId we are looking for similar movies within the bracket. Here we used 150 as an example. The print function prints the result to the screen.



```
print('Similar movies are: \n')
similar_movie_res = similar_movie(150)
print(similar_movie_res)
```

Alternatively, we can save it as a CSV file to our folder with ".to_csv()" for future usage.



```
# similar_movie_res.to_csv("Question_1b.csv")
# print("Question 1 b) saved! ")
```

Question c)

This question is an upgraded version of question (b), and based on the answer in (b), not only must at least one genre and one rating be the same, but at least one tag must also be the same. So we have two parameters while defining the function, result dataframe from (b) and movieId.

```python
def enhancing_similar_movie(df, x):
    """
    input df: a dataframe contains movies which have been filtered by genres and ratings
    input x: given movie id
    return: a dataframe contains movies which also been filtered by tags
    """
```

As in the last question, target tags are obtained as the way we got ratings. And a new dataframe is created as well.

```python
tag = tags_df[tags_df['movieId'] == x].loc[:,'tag']
tag_res = pd.DataFrame(columns = ['movieId'])
```

By iterating all the rows, if the tag string is within our target values, we then append it to the result dataframe.

```python
for index_t, row_t in tags_df.iterrows():
    if (row_t['tag'] in tag.values):
        tag_res = tag_res.append({'movieId': row_t['movieId']}, ignore_index = True)
```

Inner join the result from (b) and “tag_res” we just got, we got a list of desired movies.

```python
enhance_res = pd.merge(df, tag_res, on = 'movieId', how = 'inner')
```

Once more, remove duplicate rows and input movieId, reset the index and return the result.

```python
enhance_res = enhance_res.drop_duplicates()
```

```python
enhance_res.drop(enhance_res.loc[enhance_res['movieId'] == x].index, inplace=True)
```

```python
return enhance_res.reset_index(drop=True)
```

Call the “enhance_similar_movie” function to get the final list and always remember to “similar_movie” function before since its result is one of the parameters.

```python
print('Enhancing similar movies are: \n')
enhance_similar_movie_res = enhancing_similar_movie(similar_movie_res,150)
print(enhance_similar_movie_res)
```

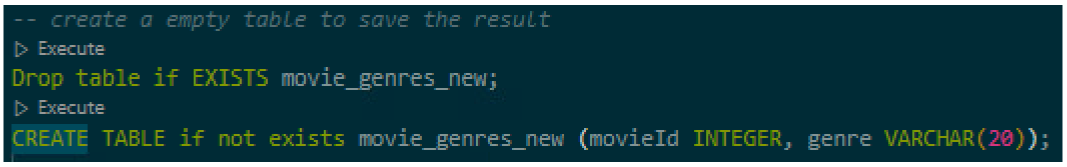
Task 2

Our goal for this task is to separate genres and save them as (movieId, genre) pairs in the new database. This will be more suitable for database processing as no more splits are needed.

The strategy for solving this task is to extract each genre from front to back. To know how many times we need to extract, we need the maximum number of genres among all the rows as the first step. By substituting the bar “|” to nothing and then calculating the difference of string length before and after the substitution, we know the maximum occurrence of bars “|” in the “genres” string is 9. The maximum number of genres is reached when we add 1, which means there are at most 10 genres waiting to be split.

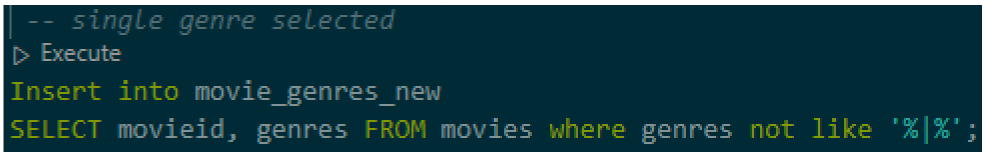
```sql
-- find out the maximum number of genres, which is max(occurrences) = 10
▷ Execute
SELECT MAX(LENGTH(genres) - length(REPLACE(genres, '|', ''))) + 1 AS max_occurrences FROM movies;
```

To continue, we create an empty table with two columns, movieId and genre, to hold the result.



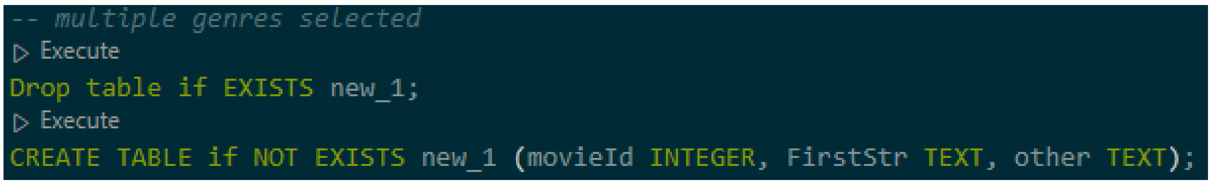
```
-- create a empty table to save the result
▷ Execute
Drop table if EXISTS movie_genres_new;
▷ Execute
CREATE TABLE if not exists movie_genres_new (movieId INTEGER, genre VARCHAR(20));
```

The entire database is split into two sections; movieId has just one genre, while the others have several. We only need to filter out and save the movies that fit into a single genre instead of splitting those movies into different genres. We can determine whether a genre is single or not by looking for any "|" symbols in the genre string with “like ‘%|%’ ”.



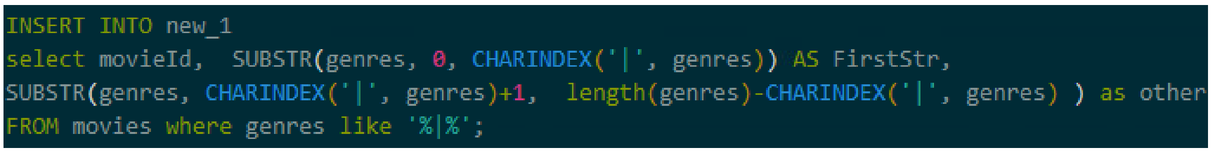
```
-- single genre selected
▷ Execute
Insert into movie_genres_new
SELECT movieid, genres FROM movies where genres not like '%|%';
```

We are now focused on handling multiple genres. Each step begins with the creation of a new table “new_n” with three columns: “movieId”, “nthStr”, and “other”, where n is the nth table created. There are at most 10 genres in the string, we need to create 9 tables during this process since the last genre will be in the other column of the ninth table.



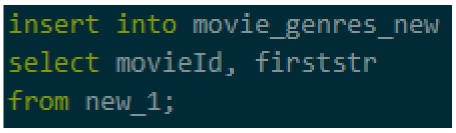
```
-- multiple genres selected
▷ Execute
Drop table if EXISTS new_1;
▷ Execute
CREATE TABLE if NOT EXISTS new_1 (movieId INTEGER, FirstStr TEXT, other TEXT);
```

Each genre is divided here using the "SUBSTR()" function, which is used to extract a portion of a string. It will always begin with the first character in the string and end before it encounters the first "|". To know the index of “|”, “CHARINDEX()” is used to help. The genre is then split in this way and saved into the “nthStr” column. The remainder of the string is then obtained using "SUBSTR()" and "CHARINDEX()" once more, and it is saved as "other."



```
INSERT INTO new_1
select movieId,  SUBSTR(genres, 0, CHARINDEX('|', genres)) AS FirstStr,
SUBSTR(genres, CHARINDEX('|', genres)+1,  length(genres)-CHARINDEX('|', genres) ) as other
FROM movies where genres like '%|%';
```

Consequently, we save the split genre, which is “FirstStr” in the first step, associated with the corresponding movieId to the result table.



```
insert into movie_genres_new
select movieId, firststr
from new_1;
```

In the meantime, at this step, there will be one genre in the “other” column for those who have two genres in all. These genres can no longer be split and can be saved into the result table as well. We use “like ‘%|%’ ” again to decide. This reduces running time, saves memory and eliminates the need to remove blanks later.

```
insert into movie_genres_new
select movieId, other
from new_1 where other not like '%|%';
```

Keep repeating these four steps until table “new_9”, where the “NinthStr” column contains the ninth genre if it exists and the “last” column contains the rest of the genres, which is the tenth genre. All of the genres are divided and saved in this way.

Task 3

In this question, we will do the same thing as in tasks 1(a)&(b) – find similar movies, but in SQL instead of python.

Question a)

The main output is SQL, but Python is used to create a function that allows movieId to be passed as parameters. We initially import pandasql to compile SQL in Python and three files: movies, ratings, and genres, which is attained in task 2.

```
import pandas as pd
from pandasql import sqldf
```

```
movies = pd.DataFrame(pd.read_csv("movies.csv"))
genres = pd.DataFrame(pd.read_csv("genres.csv"))
ratings = pd.DataFrame(pd.read_csv("ratings.csv"))
```

As expected, we create a function like 1(b). In order to distinguish between two functions, we name this function “similar_movie_sql”.

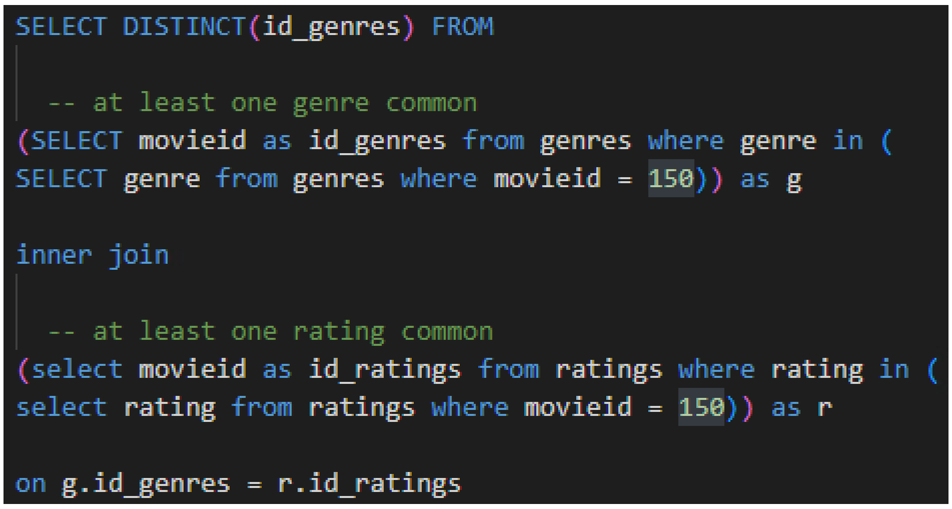
```
def similar_movie_sql(id):
```

Subsequently, we come to the SQL clause. Select the genres of the given movie ID first, then all the movies whose genres fall within. We name this list g short for genre and the filtered movieId is called "id_genres". In the same way, we have a list named r that has a column called “id_ratings”.

```
  -- at least one genre common
(SELECT movieid as id_genres from genres where genre in (
SELECT genre from genres where movieid = 150)) as g
```

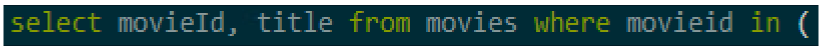
```
  -- at least one rating common
(select movieid as id_ratings from ratings where rating in (
select rating from ratings where movieid = 150)) as r
```

After getting this two separately, we inner joined them on the same column, movieId. Since there are two columns altogether and they are all the same, we choose one of them to continue. The column now contains lots of duplication and we apply “DISTINCT()” to return only unique values.

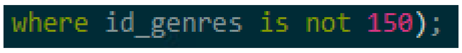


```sql
SELECT DISTINCT(id_genres) FROM

  -- at least one genre common
(SELECT movieid as id_genres from genres where genre in (
SELECT genre from genres where movieid = 150)) as g

inner join

  -- at least one rating common
(select movieid as id_ratings from ratings where rating in (
select rating from ratings where movieid = 150)) as r

on g.id_genres = r.id_ratings
```

At the end, we gathered information about movies which is one of the distinct Ids and removed the search id itself just before the result was returned.

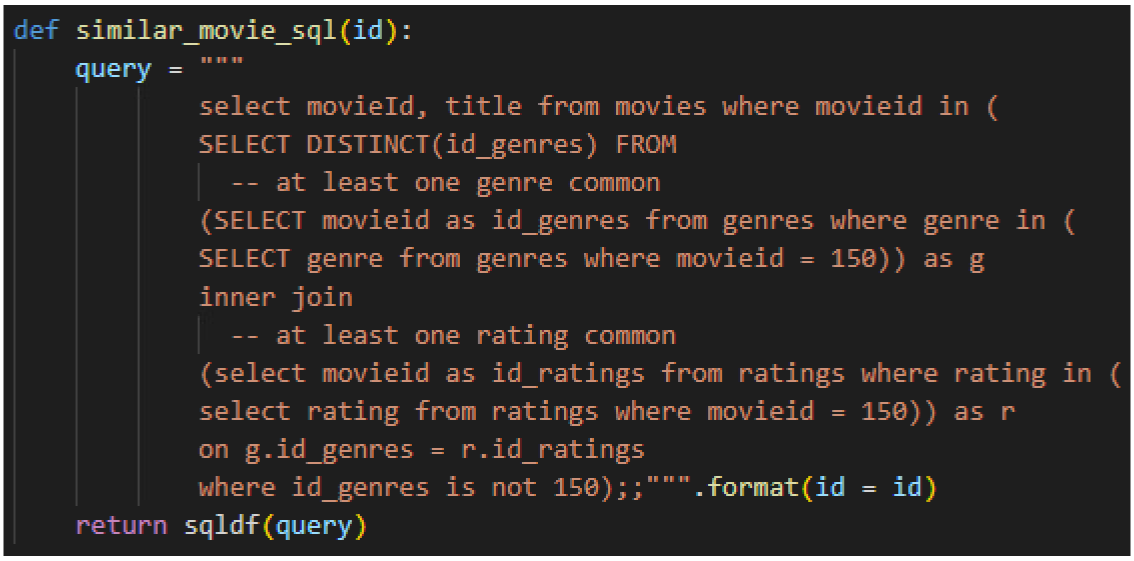


```sql
select movieId, title from movies where movieid in (
```



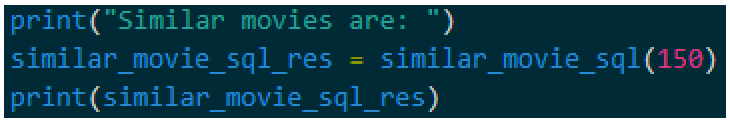
```sql
where id_genres is not 150);
```

To put input in sql, we do this in python with string formatting.



```python
def similar_movie_sql(id):
    query = """
            select movieId, title from movies where movieid in (
            SELECT DISTINCT(id_genres) FROM
              -- at least one genre common
            (SELECT movieid as id_genres from genres where genre in (
            SELECT genre from genres where movieid = 150)) as g
            inner join
              -- at least one rating common
            (select movieid as id_ratings from ratings where rating in (
            select rating from ratings where movieid = 150)) as r
            on g.id_genres = r.id_ratings
            where id_genres is not 150);;""".format(id = id)
    return sqldf(query)
```

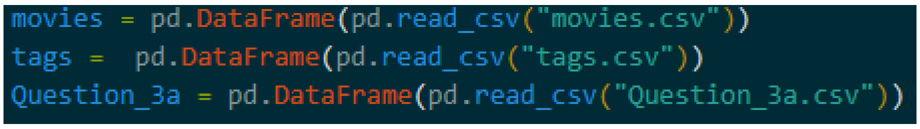
Finally we call the function and print it out.



```python
print("Similar movies are: ")
similar_movie_sql_res = similar_movie_sql(150)
print(similar_movie_sql_res)
```

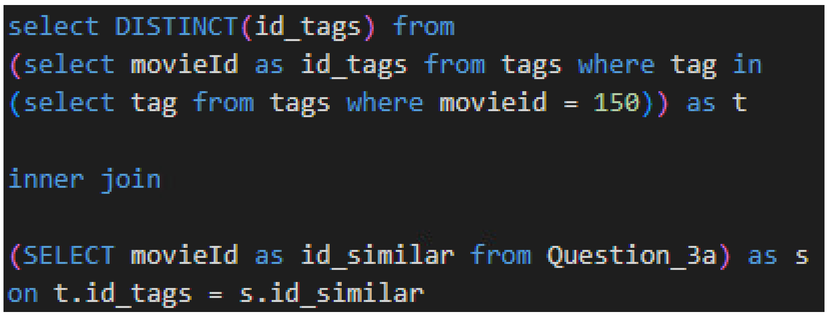
Question b)

Since the result list from last 3(a) is needed to perform 3(b), we already saved it as “Question_3a.csv” in the last step. Import the required CSV file and the file we got from last step, we then finished importing.



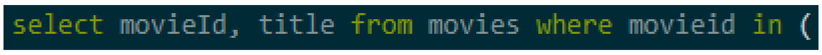
```
movies = pd.DataFrame(pd.read_csv("movies.csv"))
tags =  pd.DataFrame(pd.read_csv("tags.csv"))
Question_3a = pd.DataFrame(pd.read_csv("Question_3a.csv"))
```

As in the previous step, we select tags with movieIds of 150 and movies with the same tag.Remove duplicates by combining two parts with an inner join on the same column.

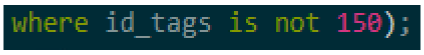


```
select DISTINCT(id_tags) from
(select movieId as id_tags from tags where tag in
(select tag from tags where movieid = 150)) as t

inner join

(SELECT movieId as id_similar from Question_3a) as s
on t.id_tags = s.id_similar
```

Finally, the movies that match the search criteria are returned, but the search id movie is removed.

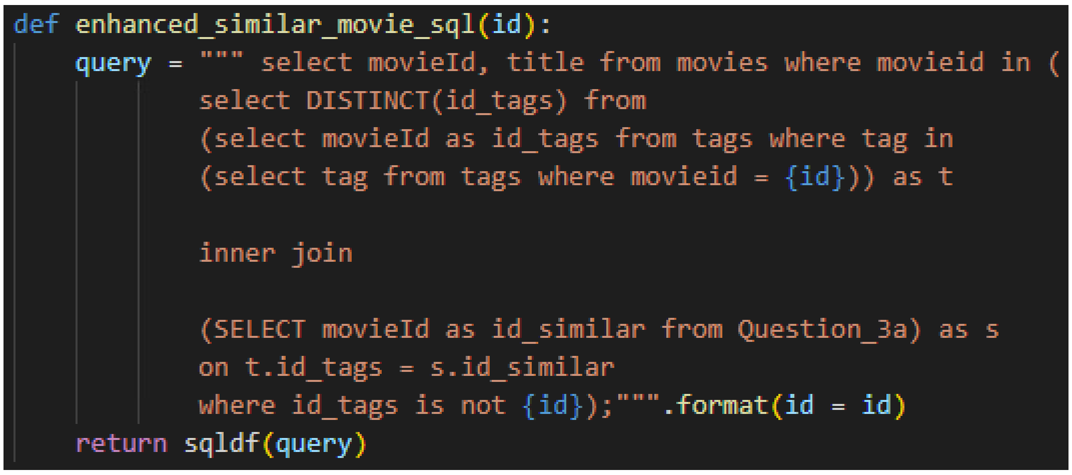


```
select movieId, title from movies where movieid in (
```

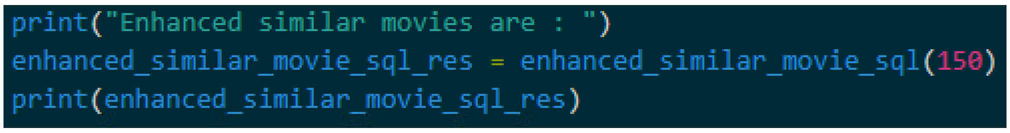


```
where id_tags is not 150);
```

Then, in Python, putting input is accomplished through string formatting.



```
def enhanced_similar_movie_sql(id):
    query = """ select movieId, title from movies where movieid in (
            select DISTINCT(id_tags) from
            (select movieId as id_tags from tags where tag in
            (select tag from tags where movieid = {id})) as t

            inner join

            (SELECT movieId as id_similar from Question_3a) as s
            on t.id_tags = s.id_similar
            where id_tags is not {id});""".format(id = id)
    return sqldf(query)
```



```
print("Enhanced similar movies are : ")
enhanced_similar_movie_sql_res = enhanced_similar_movie_sql(150)
print(enhanced_similar_movie_sql_res)
```

Task 4

In this task, we imported all the files, including three given files and five result files from the previous question. The whole strategy we are using here is comparing the actual result with the expected result. If they are the same, then this test is passed; otherwise, this test is failed. We decide to test in different input, which is different movieId, for each question and try to make the test as comprehensive as possible, which means including all scenarios as much as possible. For example, we perform 4 tests on task1(b), looking for similar movies, to make sure this function will work under different cases. For each test case, we will use SQL to help find the input ids that meet our conditions if it is required. After going through all the test cases, all of them are successful.