CSCI.620.01/03

Homework 3 Report (ss7495)Suraj Sureshkumar

Question1

- Used the generated csv and tsv files from assignment 2 to create a new relation as mentioned in the description of the question 1.
- Pandas and numpy were used for cleaning and generating the new data.
- Created one dataframe and read a file which had the common primary key column(foreign key in other table columns).
- Replacing the \N with NaN value.
- Used columns which were necessary and removed columns which were not so useful.
- Used inner join to join the columns and mentioned on what to join.
- Took movies which had runtimeMinutes greater than or equal to 90 and created the csv file by joining the rest of the columns.
- With the help of the first generated file implemented a code to filter out movies where all actors only play a single character.

Please find the source code in the file name sourcecode.zip

To run the code files you will need to run the main.py first and then the remove_character.py. The main.py will give you a file with the name new_relation.csv. Keep in mind that since I have used suffixes, additional columns might be added which I am dropping before creating the final csv file. In case you encounter any additional suffixes(you most sure will not) please drop it. Now use the new_relation.csv for the program 2 which will give a tsv file.

The code comment for the first program is in doc string format due to the pycharm on my machine gave error for single line comments

The below code has been just for reference purpose.

main.py:

```
import numpy as np
import pandas as pd

tsv_file1 = "title.csv"
tsv_file2 = "movie_genre_mapping.csv"
tsv_file3 = 'title_actor.tsv'
tsv_file4 = 'member.csv'
tsv_file5 = 'title_actor_character.csv'
tsv_file6 = 'genre_movie_genre_id.csv'
tsv_file7 = 'characters.tsv'

df1 = pd.read_csv(tsv_file1, usecols=["tconst", "titleType", "startYear",
"runtimeMinutes", "averageRating"])

df1.replace(r'\N', np.NaN, inplace=True)
df1.dropna(inplace=True)
df1 = df1[df1.runtimeMinutes.astype(int) >= 90]
df1 = df1.join(pd.read_csv(tsv_file2), on='tconst', how='inner', lsuffix='',
rsuffix='_right') \
    .join(pd.read_table(tsv_file3, sep='\t'), on='tconst', how='inner', lsuffix='',
rsuffix='_right2') \
    .join(pd.read_csv(tsv_file4, usecols=['nconst', 'birthYear']), on='nconst',
how='inner', lsuffix='',
          rsuffix='_right3') \
    .join(pd.read_csv(tsv_file5), on='tconst', how='inner', lsuffix='',
          rsuffix='_right4')\
    .join(pd.read_table(tsv_file7), on='id',how='inner', lsuffix='',
          rsuffix='_right5')
df1 = df1.join(pd.read_csv(tsv_file6), on='tconst', how='inner', lsuffix='',
              rsuffix='_right5')

df1.to_csv('new_relation.csv', index=False)
```

remove_character.py:

```
import pandas as pd

df = pd.read_csv('new_relation.csv')

vals = df[['nconst', 'id']].value_counts()  # counting values nconst unique values

nconst = [i[0] for i in vals[vals.values > 1].index]  # seaprating value of nconst which are greater than 1
# taking ids which are greater than 1( which is repeated more than once nconst and id together)
ids = [i[1] for i in vals[vals.values > 1].index]


def new_col(row):
    """
    Passing each and every row of dataframe
    :param row:
    :return:
    """
    if row['nconst'] in nconst:  # checking if row nconst in nconst list
        idx = nconst.index(row['nconst'])  # if present checking the index of the nconst
        # if that index value is in row id then returning 0 determining duplicate or else return 1
        if ids[idx] == row['id']:
            return 0
    return 1


df['col'] = df.apply(new_col, axis=1)  # got 0 and 1s in new columns

remove_movies = df[
```

```
    df.col == 0].tconst.values  # taking columns which have 0 value and the tconst value and storing in remove_movies

df['col'] = df.tconst.apply(
    lambda x: 0 if x in remove_movies else 1)  # if row value in remove_movies column is assigned 0 or else 1
df = df[df.col == 1]  # then columns which have 1
df.drop('col', axis=1).to_csv(find_dependencies.tsv, index=False)
```

Representation of how the table looks after uploading it to the database

	movieId	type	startYear	runtimeMinutes	averageRating	genres	memberId	birthYear	genre_id	characters
1	147	movie	1897	100	5	Short	151209	<null>	1	0
2	502	movie	1905	100	4	Short	152323	<null>	1	0
3	591	movie	1907	90	4	Short	201558	<null>	1	0
4	2445	movie	1913	120	6	Drama	85066	<null>	1	0
5	2646	movie	1913	121	6	Drama	334918	<null>	7	0
6	2885	movie	1913	90	6	Western	1396754	<null>	1	0
7	3165	movie	1913	90	6	Western	128633	<null>	1	0
8	3241	movie	1913	90	5	\N	102858	<null>	1	0
9	3740	movie	1914	148	7	Drama	845505	1899	1	0
10	4052	movie	1914	1428	6	Comedy	3068222	<null>	7	0
11	4465	movie	1914	199	6	Short	1495634	<null>	7	0
12	4525	movie	1914	109	5	Horror	275943	1963	7	0
13	8326250	movie	2018	104	5	Drama	125164	<null>	9	92858
14	4974	movie	1915	195	2	Short	149600	1945	1	0
15	6914	movie	1916	93	5	Short	686142	<null>	3	0
16	7684	movie	1917	100	5	Drama	346808	<null>	1	0
17	8144	movie	1917	100	4	Drama	583645	<null>	4	0
18	8150	movie	1916	138	6	Drama	1363590	<null>	1	0
19	8395	movie	1918	112	6	Horror	1073465	<null>	1	0
20	8662	movie	1917	90	5	\N	316947	<null>	7	0
21	9014	movie	1918	94	6	Short	713360	<null>	2	0
22	9359	movie	1917	92	5	Comedy	337033	<null>	7	0
23	9369	movie	1918	93	6	Drama	669895	<null>	7	0
24	6390434	tvMovie	2016	90	8	Drama	124279	<null>	7	39152

Question2:

Functional Dependencies for the tables are:

1. memberId→ birthYear

Each member will have a birthYear and by using the memberId we can find the birthYear of the respective member.

2. movieId → type

Each movie will have its own type and there can be only one type for a given movie.

3. movieId → startYear

Each movie will have its own unique startYear compared to other movies.

4. genreId → genre

Each genreId has its own genre associated with it and hence this is a functional dependency.

5. movieId → runtimeMinutes

Each movie will have its one unique runtimeMinutes with it.

6. movieId → avgRating

Each movie will have its own rating and a movie can have one avgRating.

7. movieId, memberId → character

A combination of movieId and memberId will give us the character associated with that movie and with that memberId.

8. genre → genreId

Each genre has its own genreId associated with it and hence this is a functional dependency.

Question3:

The code file is named find_functional_dep.py. To run the program add the final file generated by running the program 1 first and then the program 2. Program 2 will give you a tsv file. Make sure that all modules are installed and run.

My program took a total time of **1 hr 28 mins 33 secs** to complete.

The program I implemented used a column by column approach to find functional dependencies.

The flow of the program is like this:
Read the file to the dataframe and apply a loop to loop for the columns and then one more loop to loop till the length of the columns in the dataframe and one more loop to get the combinations of the columns which takes up to the length of the value in the previous loop. A if condition in the loop to check if the first loop column in the combination loop. If present then we continue cause we don't want to add it to the cols variable which is created after this. Then created a variable named cols and stored the list of combinations columns into it so that we can access only the columns. Then appended the column from first for loop to the cols variable cause we know that it is unique.
Initialized separate variables and created a set and removed the duplicates from the dataframe column and the columns which were stored in a list. Then compared the length of the two initialized variables and if they are equal then we know that they are functional dependencies and we print the output.
The code in the code file is also filled with comments explaining each step in more details.

Output produced by the program:

```
memberid --> ('birthYear',)
movieid --> ('type',)
movieid --> ('startYear',)
movieid --> ('runtime',)
movieid --> ('avgRating',)
movieid --> ('type', 'startYear')
movieid --> ('type', 'runtime')
movieid --> ('type', 'avgRating')
movieid --> ('startYear', 'runtime')
movieid --> ('startYear', 'avgRating')
movieid --> ('runtime', 'avgRating')
movieid --> ('type', 'startYear', 'runtime')
movieid --> ('type', 'startYear', 'avgRating')
movieid --> ('type', 'runtime', 'avgRating')
movieid --> ('startYear', 'runtime', 'avgRating')
movieid --> ('type', 'startYear', 'runtime', 'avgRating')
genre --> ('genre_id',)
genre_id --> ('genre',)

Process finished with exit code 0
```

Question4:

The functional dependencies from question 2 and 3 overlap. But, for some cases there were some functional dependencies which were repeated. As my initial list of dependencies included:
memberId→ birthYear
movieId → type
movieId → startYear
genreId → genre
movieId → runtimeMinutes
movieId → avgRating
movieId, memberId → character
genre → genreId

The functional dependencies like memberId, movieId → character were not included because the question stated to restrict to one attribute on the left hand side. So the program I implemented took one column on the left hand side rather than two or more to check multi dependencies.
When you look at the output image below you can see a some of them are redundant

```
memberid --> ('birthYear',)
movieid --> ('type',)
movieid --> ('startYear',)
movieid --> ('runtime',)
movieid --> ('avgRating',)
movieid --> ('type', 'startYear')
movieid --> ('type', 'runtime')
movieid --> ('type', 'avgRating')
movieid --> ('startYear', 'runtime')
movieid --> ('startYear', 'avgRating')
movieid --> ('runtime', 'avgRating')
movieid --> ('type', 'startYear', 'runtime')
movieid --> ('type', 'startYear', 'avgRating')
movieid --> ('type', 'runtime', 'avgRating')
movieid --> ('startYear', 'runtime', 'avgRating')
movieid --> ('type', 'startYear', 'runtime', 'avgRating')
genre --> ('genre_id',)
genre_id --> ('genre',)

Process finished with exit code 0
```

Dependencies like movieId →(‘type’,’runtime’), movieId → (‘type’,’startYear’,’runtime’) are redundant cause we already got the dependency from of single dependency and were not included:
movieId → type
movieId → startYear
movieId → runtimeMinutes
So combining these three into one will give us the above dependency which is redundant and need not be mentioned again. Because, the functional dependencies were found to prove that unique data can be found.

Question5:

3NF decomposition:
The table is in 1NF as the question 1 stated to remove movies where an actor plays more than one role. 2NF has also been resolved as relations with a primary key composed of two or more attributes were resolved as they are known as transitive dependencies. The below relations signify 3NF decomposition:

So the 3NF decomposition(final decomposition) is:
(memberId, movieId, character)

(memberId, movieId, genreId)

(genreId, genre)

(memberId, birthYear)

(movieId, type, startYear, runtimeMinutes, avgRating)

Candidate Key:

Core - (memberId, movieId, genreId)

Exterior - character, birthYear, runtimeMinutes, avgRating, type, startYear, genre.

As mentioned in the lecture slides “Test the closure of the core. If this is the complete set of attributes, the core is the only candidate key”. Hence core is the only candidate key.

Canonical cover:
The canonical cover is the reduction of the functional dependencies where redundancy is removed.

memberId → birthYear

movieId → type

movieId → startYear

movieId → runtimeMinutes

movieId → avgRating

movieId, memberId → character

genreId → genre