MovieLens Preprocessing

January 4, 2022

Francisco Javier Sáez Maldonado, José Antonio Álvarez Ocete

1 Introduction

In this assignment, we are asked to create a **data pipeline** using two csv files taken from MovieLens.

Our goal is to use the available raw data to provide a clean, valid and transformed data that helps the data scientist to obtain information about the data easily. We are told that, after our data pipeline, the following query will be done:

> **Obtain a list of movie genres, sorted by their average score obtained in the last week**.

We will orient our code to make this query as easy as possible. We will assume that our data comes in two `.csv` files: `input/movies.csv` and `input/ratings.csv`. We will perform the preprocessing of the data and it will be stored in the folder `output`.

Firstly, we will import the needed libraries. `Pandas` will help us to load the *.csv* files and to do the transformations in the data.

[1]:
```
# Imports cell
import pandas as pd
import numpy as np
from sklearn.pipeline import Pipeline
import time
from sklearn.preprocessing import FunctionTransformer
```

Now, we can read the data from the provided *.csv* files and show the first 5 elements of each file, taking a sneak peek at the data.

[2]:
```
movies = pd.read_csv("input/movies.csv")
ratings = pd.read_csv("input/ratings.csv")
```

[3]:
```
ratings.head(5)
```

[3]:
```
   userId  movieId  rating   timestamp
0       1      296     5.0  1147880044
1       1      306     3.5  1147868817
2       1      307     5.0  1147868828
3       1      665     5.0  1147878820
```

```
4       1      899     3.5  1147868510
```

[4]:
```
movies.head(5)
```

[4]:
```
   movieId                               title  \
0        1                    Toy Story (1995)
1        2                      Jumanji (1995)
2        3             Grumpier Old Men (1995)
3        4            Waiting to Exhale (1995)
4        5  Father of the Bride Part II (1995)

                                        genres
0  Adventure|Animation|Children|Comedy|Fantasy
1                   Adventure|Children|Fantasy
2                               Comedy|Romance
3                         Comedy|Drama|Romance
4                                       Comedy
```

As we can see, the data contained in each file is quite different, so we will analyze each part separately and we will try to combine their information at the end to produce the output desired in our query.

As a first comment, we must see that in this case the concept of **outliers** does not match to our data.

- In the `ratings` dataframe, the *outliers* will be the incorrect ratings or negative times.
- In the `movies` dataframe, our outliers will be the invalid genres.

Also, **no normalization** is needed in this case. The only possible case of normalization would be to crop the wrong rating values to our interval, but we will take a different approach that will be explained further in this assignment.

2 Ratings

We will begin transforming the input data from this file. Using this data, our goal will be to clean and transform it so that obtaining the rating of each film is as easy as possible.

The procedure that we will follow is the is the next one:

1. Ignore all the data that was obtained earlier than 1 week before, since we are only interested in the data of the last week.
2. Remove all null and wrong values that we can find in the reduced table.
3. Group values by movieId and compute mean rating for each film, storing it on a table.

We begin by removing all the data that was introduced a n_weeks ago. Our goal is to use only the last week ($n\_weeks = 1$), but since the data last updated more than 2 years ago, we code a general function that takes the data from the last n_weeks and stores it in a dataframe. Using n_weeks, we are **assuming that the processed data will only be used to perform this query and not other queries**.

In the code example, we will use 120 weeks, although in the final pipeline a single week will be used. This is done with correctness-showing purposes

```
[5]: week = 60*60*24*7
     filter_date = lambda df, n_weeks = 1 : df[df['timestamp'] > time.time() -␣
      ↪n_weeks *week ]
```

```
[6]: # Remove invalid dates
     print("Nº ratings before removing invalid dates: {}".format(len(ratings.index)))
     ratings_date = filter_date(ratings, 120)
     print("Nº ratings after removing invalid dates: {}".format(len(ratings_date.
      ↪index)))
```

```
Nº ratings before removing invalid dates: 25000095
Nº ratings after removing invalid dates: 259555
```

We have already removed the data that, by timedate, is not usefull for us. Now, we have to remove the incorrect/wrong data. The most basic step is to remove the rows of the dataframe that contain null values and also the columns that are duplicated. We combine this two actions in one function.

```
[7]: # Function that removes rows that have a null value
     remove_nulls = lambda df : df.dropna().drop_duplicates()
```

```
[8]: # Remove nulls
     print("Nº ratings before removing: {}".format(len(ratings_date.index)))
     ratings_no_nulls = remove_nulls(ratings_date)
     print("Nº ratings after removing: {}".format(len(ratings_no_nulls.index)))
```

```
Nº ratings before removing: 259555
Nº ratings after removing: 254432
```

Having removed all the completely invalid elements of our data, we want to remove values that may seem to be correct but are not in the subset of possible values: $\{0.5 \cdot i\}_{i=1}^{10}$ (we assume that the ratings are in this set because the documentation says so). A few approaches can taken in this spot, making some assumptions in the current values of the table. For instance, we could assume that all the values hat have a negative sign (which, we have checked that they exist), either are the same values but with a positive sign or are the minimum score. Since making an interpretation on this is complicated, we decide to simply eliminate them so that we do not introduce a bias in the dataset.

```
[9]: def remove_wrong_evaluations(df):
         correct_vals = np.arange(0.5, 5.01, 0.5)
         return df[df['rating'].isin(correct_vals)]
```

```
[10]: print("Nº ratings before removing wrong: {}".format(len(ratings_no_nulls.
       ↪index)))
      ratings_no_wrong = remove_wrong_evaluations(ratings_no_nulls)
      print("Nº ratings after removing wrong: {}".format(len(ratings_no_wrong.index)))
```

```
Nº ratings before removing wrong: 254432
Nº ratings after removing wrong: 254338
```

We have removed all the meaningless data. Now, our goal is to create a field on this database that will help us perform the previously mentioned query much easily. We want to create a new column in this dataframe that indicates the **average rating** of each movie in the last week.

We only have to group all the elements of the final dataframe by the field `movieId`, and then compute the mean of the field `rating` for each group obtained in the grouping stage.

[11]:
```
mean_per_id = lambda df: df.groupby('movieId')['rating'].mean()
```

[12]:
```
ratings_store = mean_per_id(ratings_no_wrong)
```

[13]:
```
ratings_store.head(5)
```

[13]:
```
movieId
0    3.541176
1    3.984507
2    3.541667
3    3.300000
4    0.500000
Name: rating, dtype: float64
```

As we can see, we obtain a `pandas.Series` where we have the average rating that each film obtained in the last *n_weeks*. Some films may have dissapeared, but this is **not a problem**, since if we wanted them to be in this dataframe, their score would have to be the lowest possible, and this would introduce a bias in the average score of the gender of the film.

2.1 Final Pipeline

Lastly, to make the code cleaner and alike-looking to a *Pipeline*, we are going to use `sklearn.Pipeline` to compactify this process. We previously declare a lambda function that saves the dataframe to a *csv*

[14]:
```
save = lambda df,name : df.to_csv(name, index = False)
```

[15]:
```
average_rating_pipeline = Pipeline([
                            ('date_filter',FunctionTransformer( func =␣
 ↪filter_date, kw_args = {'n_weeks': 120})),
                            ('remove_nulls',FunctionTransformer(func =␣
 ↪remove_nulls)),
                           ␣
 ↪('remove_wrong_evaluations',FunctionTransformer(func =␣
 ↪remove_wrong_evaluations)),
                            ('compute_mean_id',FunctionTransformer(func =␣
 ↪mean_per_id)),
                            ('save',FunctionTransformer( func = save, kw_args =␣
 ↪{'name': "output/avg_movie.csv"}))
```

```
                                   ])
```

[16]:
```
transformed_data = pd.read_csv("output/avg_movie.csv")
```

[17]:
```
transformed_data.head(5)
```

[17]:
```
   movieId    rating
0        0  3.541176
1        1  3.984507
2        2  3.541667
3        3  3.300000
4        4  0.500000
```

As we can see, we have obtained the same result in both executions, so from now on we will only use the `average_rating_pipeline` transformer.

2.2 About the removed data

Let us present in tables the information that we have already shown in the previous steps.

In our case, considering the % eliminated by date is pointless, since we already know that the data stopped updating almost 2 years ago. We resume it in the following table.

Transformation	Before	After	% kept
Invalid date	25000095	259911	1%
Null/duplicated	259911	254783	98%
Invalid rating	254783	254689	99.9%

Recall that **all this statistics are referred to the data of the last** 120 **weeks, since this is the subset we are considering**.

3 Movies

It is now time to preprocess the `movies.csv` information. In this case, the preprocessing will be different.

Firstly, we already know (from the documentation) that the `genres` of the films are all valid but those one listed as `(no genres listed)`, so we will have to remove those and any other null values.

Secondly, the genres are provided in a `string`, where the character `|` separates the different genres of the same film. Our goal in stage will be to split the films that contain k genres into k entries in the table. Each new element will have the same `movieId` but will have a unique genre (one new entry for each of the k genres).

[18]:
```
# Remove nulls
def remove_null_genres(df):
    return df[df.genres != '(no genres listed)']
```

```
[19]: print("Movies before removing invalid: {}".format(len(movies.index)))
movies_no_null = remove_null_genres(movies)
print("Movies after removing null genres: {}".format(len(movies_no_null.index)))
movies_no_null = remove_nulls(movies_no_null)
print("Movies after removing invalid entries: {}".format(len(movies_no_null.
 ↪index)))
```

```
Movies before removing invalid: 62423
Movies after removing null genres: 57361
Movies after removing invalid entries: 57361
```

After removing the null or invalid genres, we have to split the `genre` column into the different genres creating new rows for the dataframe.

```
[20]: def split_genres(df):
    df['genres'] = df['genres'].apply(lambda x : x.split("|"))
    return df.explode('genres')
```

```
[21]: print("Movies shape before split genres : {}".format(movies_no_null.shape))
df_split = split_genres(movies_no_null)
print("Movies shape after split genres: {}".format(df_split.shape))
```

```
Movies shape before split genres : (57361, 3)
Movies shape after split genres: (107245, 3)
```

We can appretiate how several columns have been added. We can also show this in the first 3 elements:

```
[22]: df_split.head(3)
```

```
[22]:    movieId               title     genres
0          1   Toy Story (1995)  Adventure
0          1   Toy Story (1995)  Animation
0          1   Toy Story (1995)   Children
```

3.1 Final pipeline

As we did before, we can create a `sklearn.Pipeline` to compactify the process.

```
[23]: movie_genre_pipe = Pipeline([
                            ('date_filter',FunctionTransformer( func =␣
 ↪remove_null_genres)),
                            ('remove_nulls',FunctionTransformer(func =␣
 ↪remove_nulls)),
                            ('split_genres',FunctionTransformer(func =␣
 ↪split_genres)),
                            ('save',FunctionTransformer( func = save, kw_args =␣
 ↪{'name': "output/split_genre.csv"})),
])
```

[24]:
```
movie_genre_pipe.transform(movies)
```

[25]:
```
split_pipe = pd.read_csv("output/split_genre.csv")
```

[26]:
```
split_pipe.head(5)
```

[26]:
```
   movieId             title     genres
0        1  Toy Story (1995)  Adventure
1        1  Toy Story (1995)  Animation
2        1  Toy Story (1995)   Children
3        1  Toy Story (1995)     Comedy
4        1  Toy Story (1995)    Fantasy
```

3.2 About the removed data

As we can see in the outputs of the code, there are a few changes in the size of our data. We can resume it in the following table.

Transformation	Before	After	% change
Invalid genres	62423	57361	0.91%
Null/duplicated	57361	57361	0%
Split genres	57361	107245	186.9 %

There is not a big change when removing invalid genres and no change when remove null or duplicated rows. The big change here comes when we split the genres from a single row for each movie to a row for each genre of each movie.

4 Dealing with the new data

We assume that, every day at 3.00A.M., new data will be available for us in the `input` folder. At this time, the data must be re-processed using the code that we have developed in this notebook. For an easier application of this code, we have created a script called `preprocessing.py`, that encapsulates all the preprocessing that it is done in this notebook.

We will now explain how to perform the task daily in a **Linux environment**.

To execute the preprocessing each day when the new data comes, we have to use `Cron` to schedule a daily task at that time that executes this script. As an example of the `cron` file that we would have to declare in the `/etc/cron.d` folder, we could find:

```
# Run preprocessing.py every day at 3.00 AM
SHELL=/bin/bash
PATH=/home/user/MovieLens/

0 3 * * * "python preprocessing.py"
```

5 The query

After the data has been preprocessed, we are ready to perform the desired query to obtain the sorted list of genres by rating in the last 120 weeks (remember that we do this so that we obtain any information in our preprocessing, since the last time this data was updates was more than 2 years ago). To achieve this, we have to load the preprocessed datasets and, then, perform the query using `Pandas` transformations. Recall that all the following transformations **have an equivalent done in SQL**, supossing that each dataframe is a table.

[31]:
```
pre_movies = pd.read_csv("output/split_genre.csv")
pre_ratings = pd.read_csv("output/avg_movie.csv")

# (SQL) Join both tables and then drop movieId
df = pd.merge(pre_movies, pre_ratings, on = 'movieId').drop("movieId",axis = 1)

# Group by genre and compute mean of ratings
df = pd.DataFrame(df.groupby("genres")['rating'].mean()).
↪sort_values(by=['rating'], ascending = False)

# Rename column for convenience
df = df.rename(columns={'rating': 'Last 120 Weeks Average Rating'})
```

[32]:
```
df
```

[32]:
```
                        Last 120 Weeks Average Rating
genres
Film-Noir                                    3.597176
Psychological Thriller                       3.563435
War                                          3.451769
Documentary                                  3.428540
Animation                                    3.410993
Musical                                      3.333299
Drama                                        3.306670
Crime                                        3.262375
Western                                      3.246771
IMAX                                         3.217749
Children                                     3.212805
Romance                                      3.211524
Espionage Action                             3.193313
Mystery                                      3.186912
Adventure                                    3.178372
Fantasy                                      3.178367
Comedy                                       3.126044
Action                                       3.121551
Thriller                                     3.075556
Sci-Fi                                       3.016054
Horror                                       2.763636
```

As we can see, we make use of a *JOIN* and a *GROUPBY* SQL operations, and finally we compute the means of one of the columns obtained by the `GROUPBY` and sort the result of the means computation. This code can be independently executed using the `query.py` script.

The `query.py` is independent of the number of weeks that have been chosen to obtain the average ratings, so it could be used in an environment where we have weekly updated data, as well as the `preprocessing.py` script could be, only changing the `n_weeks` parameter.