IMDB

July 3, 2024

<div class="alert alert-success"; style="border-left: 7px solid green"> Reviewer's comment, v. 2

Great! You've done a great job and now your project has been accepted.

Thank you for your work and I wish you success in the following projects!

Hello Dhruval

My name is Dima, and I will be reviewing your project.

You will find my comments in coloured cells marked as 'Reviewer's comment'. The cell colour will vary based on the contents - I am explaining it further below.

Note: Please do not remove or change my comments - they will help me in my future reviews and will make the process smoother for both of us.

You are also very welcome to leave your comments / describe the corrections you've done / ask me questions, marking them with a different colour. You can use the example below:

<div class="alert alert-info"; style="border-left: 7px solid blue"> Student's comment

0.1 Basic Python - Project

Review summary

Dhruval, thanks for submitting the project. You've done a very good job and I enjoyed reviewing it.

- You completed all the tasks.
- Your code was optimal and easy to read.
- You wrote your own functions.

There is only one critical comment that need to be corrected. You will find it in the red-colored cell in relevant section. If you have any questions please write them when you return your project.

I'll be looking forward to getting your updated notebook.

0.2 Introduction

In this project, you will work with data from the entertainment industry. You will study a dataset with records on movies and shows. The research will focus on the "Golden Age" of television, which began in 1999 with the release of *The Sopranos* and is still ongoing.

The aim of this project is to investigate how the number of votes a title receives impacts its ratings. The assumption is that highly-rated shows (we will focus on TV shows, ignoring movies) released during the “Golden Age” of television also have the most votes.

0.2.1 Stages

Data on movies and shows is stored in the `/datasets/movies_and_shows.csv` file. There is no information about the quality of the data, so you will need to explore it before doing the analysis.

First, you’ll evaluate the quality of the data and see whether its issues are significant. Then, during data preprocessing, you will try to account for the most critical problems.

Your project will consist of three stages: 1. Data overview 2. Data preprocessing 3. Data analysis

0.3 Stage 1. Data overview

Open and explore the data.

You’ll need `pandas`, so import it.

```
[2]: # importing pandas
     import pandas as pd
```

Read the `movies_and_shows.csv` file from the `datasets` folder and save it in the `df` variable:

```
[3]: # reading the files and storing them to df
     df = pd.read_csv('/datasets/movies_and_shows.csv')
```

Print the first 10 table rows:

```
[4]: # obtaining the first 10 rows from the df table
     # hint: you can use head() and tail() in Jupyter Notebook without wrapping them␣
      ↪into print()
     print(df.head(10))
```

```
             name                        Character   r0le        TITLE   Type  \
0   Robert De Niro                   Travis Bickle  ACTOR  Taxi Driver  MOVIE
1     Jodie Foster                   Iris Steensma  ACTOR  Taxi Driver  MOVIE
2    Albert Brooks                             Tom  ACTOR  Taxi Driver  MOVIE
3    Harvey Keitel          Matthew 'Sport' Higgins  ACTOR  Taxi Driver  MOVIE
4  Cybill Shepherd                           Betsy  ACTOR  Taxi Driver  MOVIE
5      Peter Boyle                          Wizard  ACTOR  Taxi Driver  MOVIE
6   Leonard Harris       Senator Charles Palantine  ACTOR  Taxi Driver  MOVIE
7   Diahnne Abbott                 Concession Girl  ACTOR  Taxi Driver  MOVIE
8      Gino Ardito             Policeman at Rally  ACTOR  Taxi Driver  MOVIE
9  Martin Scorsese  Passenger Watching Silhouette  ACTOR  Taxi Driver  MOVIE

   release Year              genres  imdb sc0re  imdb v0tes
0          1976  ['drama', 'crime']         8.2    808582.0
1          1976  ['drama', 'crime']         8.2    808582.0
2          1976  ['drama', 'crime']         8.2    808582.0
```

```
3          1976  ['drama', 'crime']         8.2    808582.0
4          1976  ['drama', 'crime']         8.2    808582.0
5          1976  ['drama', 'crime']         8.2    808582.0
6          1976  ['drama', 'crime']         8.2    808582.0
7          1976  ['drama', 'crime']         8.2    808582.0
8          1976  ['drama', 'crime']         8.2    808582.0
9          1976  ['drama', 'crime']         8.2    808582.0
```

Obtain the general information about the table with one command:

[5]:
```
# obtaining general information about the data in df
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 85579 entries, 0 to 85578
Data columns (total 9 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0      name        85579 non-null  object
 1   Character      85579 non-null  object
 2   r0le           85579 non-null  object
 3   TITLE          85578 non-null  object
 4     Type         85579 non-null  object
 5   release Year   85579 non-null  int64
 6   genres         85579 non-null  object
 7   imdb sc0re     80970 non-null  float64
 8   imdb v0tes     80853 non-null  float64
dtypes: float64(2), int64(1), object(6)
memory usage: 5.9+ MB
```

<div class="alert alert-success"; style="border-left: 7px solid green"> Reviewer's comment, v. 1

Great - you've used a comprehensive set of methods to have a first look at the data.

The table contains nine columns. The majority store the same data type: object. The only exceptions are `'release Year'` (int64 type), `'imdb sc0re'` (float64 type) and `'imdb v0tes'` (float64 type). Scores and votes will be used in our analysis, so it's important to verify that they are present in the dataframe in the appropriate numeric format. Three columns (`'TITLE'`, `'imdb sc0re'` and `'imdb v0tes'`) have missing values.

According to the documentation: - `'name'` — actor/director's name and last name - `'Character'` — character played (for actors) - `'r0le '` — the person's contribution to the title (it can be in the capacity of either actor or director) - `'TITLE '` — title of the movie (show) - `'  Type'` — show or movie - `'release Year'` — year when movie (show) was released - `'genres'` — list of genres under which the movie (show) falls - `'imdb sc0re'` — score on IMDb - `'imdb v0tes'` — votes on IMDb

We can see three issues with the column names: 1. Some names are uppercase, while others are lowercase. 2. There are names containing whitespace. 3. A few column names have digit '0' instead of letter 'o'.

0.3.1 Conclusions

Each row in the table stores data about a movie or show. The columns can be divided into two categories: the first is about the roles held by different people who worked on the movie or show (role, name of the actor or director, and character if the row is about an actor); the second category is information about the movie or show itself (title, release year, genre, imdb figures).

It's clear that there is sufficient data to do the analysis and evaluate our assumption. However, to move forward, we need to preprocess the data.

<div class="alert alert-success"; style="border-left: 7px solid green"> Reviewer's comment, v. 1

Please note that it is highly recommended to add a conclusion / summary after each section and describe briefly your observations and / or major outcomes of the analysis.

0.4 Stage 2. Data preprocessing

Correct the formatting in the column headers and deal with the missing values. Then, check whether there are duplicates in the data.

[6]:
```
# the list of column names in the df table
df.columns
```

[6]:
```
Index(['   name', 'Character', 'r0le', 'TITLE', '  Type', 'release Year',
       'genres', 'imdb sc0re', 'imdb v0tes'],
      dtype='object')
```

Change the column names according to the rules of good style: * If the name has several words, use snake_case * All characters must be lowercase * Remove whitespace * Replace zero with letter 'o'

[7]:
```
# renaming columns
df = df.rename(columns ={'   name': 'name', 'Character': 'character',
                          'r0le': 'role', 'TITLE': 'title', '  Type': 'type',␣
 ↪'release Year': 'release_year',
                          'imdb sc0re': 'imdb_score', 'imdb v0tes': 'imdb_voted'
                         })
```

<div class="alert alert-success"; style="border-left: 7px solid green"> Reviewer's comment, v. 1

This is a good way to rename the columns.

Check the result. Print the names of the columns once more:

[8]:
```
# checking result: the list of column names
print(df.columns)
```

```
Index(['name', 'character', 'role', 'title', 'type', 'release_year', 'genres',
       'imdb_score', 'imdb_voted'],
      dtype='object')
```

0.4.1 Missing values

First, find the number of missing values in the table. To do so, combine two `pandas` methods:

[9]:

```
# calculating missing values
print(df.isna().sum())
```

```
name                0
character           0
role                0
title               1
type                0
release_year        0
genres              0
imdb_score       4609
imdb_voted       4726
dtype: int64
```

<div class="alert alert-success"; style="border-left: 7px solid green"> Reviewer's comment, v. 1

The isna() method is selected to find the missing values, it's great!

Not all missing values affect the research: the single missing value in `'title'` is not critical. The missing values in columns `'imdb_score'` and `'imdb_votes'` represent around 6% of all records (4,609 and 4,726, respectively, of the total 85,579). This could potentially affect our research. To avoid this issue, we will drop rows with missing values in the `'imdb_score'` and `'imdb_votes'` columns.

[10]:

```
# dropping rows where columns with title, scores and votes have missing values
df = df.dropna(axis = 'rows')
```

<div class="alert alert-success"; style="border-left: 7px solid green"> Reviewer's comment, v. 1

Perfect!

Make sure the table doesn't contain any more missing values. Count the missing values again.

[11]:

```
# counting missing values
print(df.isna().sum())
```

```
name            0
character       0
role            0
title           0
type            0
release_year    0
genres          0
imdb_score      0
imdb_voted      0
dtype: int64
```

0.4.2 Duplicates

Find the number of duplicate rows in the table using one command:

[12]:

```
# counting duplicate rows

print(df.duplicated().sum())
```

```
6994
```

Review the duplicate rows to determine if removing them would distort our dataset.

[13]:

```
# Produce table with duplicates (with original rows included) and review last 5␣
  ↪rows
duplicates_rows = df[df.duplicated(keep = False)]
print(duplicates_rows.tail(5))
```

```
                      name                 character   role    title   type  \
85569       Jessica Cediel           Liliana Navarro  ACTOR  Lokillo  MOVIE
85570  Javier Gardeaz?bal  Agust??n "Peluca" Ort??z  ACTOR  Lokillo  MOVIE
85571        Carla Giraldo            Valery Reinoso  ACTOR  Lokillo  MOVIE
85572  Ana Mar??a S?nchez                   Lourdes  ACTOR  Lokillo  MOVIE
85577         Isabel Gaona                    Cacica  ACTOR  Lokillo  MOVIE

       release_year      genres  imdb_score  imdb_voted
85569          2021  ['comedy']         3.8        68.0
85570          2021  ['comedy']         3.8        68.0
85571          2021  ['comedy']         3.8        68.0
85572          2021  ['comedy']         3.8        68.0
85577          2021  ['comedy']         3.8        68.0
```

There are two clear duplicates in the printed rows. We can safely remove them. Call the `pandas` method for getting rid of duplicate rows:

[14]:

```
# removing duplicate rows
df = df.drop_duplicates().reset_index(drop=True)
```

Check for duplicate rows once more to make sure you have removed all of them:

[15]:

```
# checking for duplicates
print(df.duplicated())
print(df.duplicated().sum())
```

```
0        False
1        False
2        False
3        False
4        False
         …
73854    False
```

```
73855    False
73856    False
73857    False
73858    False
Length: 73859, dtype: bool
0
```

<div class="alert alert-success"; style="border-left: 7px solid green"> Reviewer's comment, v. 1

Great, you found and removed the duplicates. And did very thorough checks to make sure the duplicates are gone.

Now get rid of implicit duplicates in the `'type'` column. For example, the string `'SHOW'` can be written in different ways. These kinds of errors will also affect the result.

Print a list of unique `'type'` names, sorted in alphabetical order. To do so: * Retrieve the intended dataframe column * Apply a sorting method to it * For the sorted column, call the method that will return all unique column values

[36]:

```
# viewing unique type names
print(df['type'].sort_values().unique())
```

```
['MOVIE' 'movies' 'show' 'the movie']
```

<div class="alert alert-danger"; style="border-left: 7px solid red"> Reviewer's comment, v. 1

Please note, that according to the technical task it was asked:

- For the **sorted** column, call the method that will return all unique column values

So, we should use sorting here

<div class="alert alert-success"; style="border-left: 7px solid green"> Reviewer's comment, v. 2

Now it's perfect!

Look through the list to find implicit duplicates of `'show'` (`'movie'` duplicates will be ignored since the assumption is about shows). These could be names written incorrectly or alternative names of the same genre.

You will see the following implicit duplicates: * `'shows'` * `'SHOW'` * `'tv show'` * `'tv shows'` * `'tv series'` * `'tv'`

To get rid of them, declare the function `replace_wrong_show()` with two parameters: * `wrong_shows_list=` — the list of duplicates * `correct_show=` — the string with the correct value

The function should correct the names in the `'type'` column from the `df` table (i.e., replace each value from the `wrong_shows_list` list with the value in `correct_show`).

[22]:

```
# function for replacing implicit duplicates

def replace_wrong_show(wrong_shows_list, correct_show):
    df['type'] = df['type'].replace(wrong_shows_list, correct_show)
```

Call `replace_wrong_show()` and pass it arguments so that it clears implicit duplicates and replaces them with `SHOW`:

[23]:

```
# removing implicit duplicates
wrong_shows_list = ['shows', 'SHOW', 'tv show', 'tv shows', 'tv series', 'tv']
correct_show = 'show'
replace_wrong_show(wrong_shows_list, correct_show)
```

<div class="alert alert-success"; style="border-left: 7px solid green"> Reviewer's comment, v. 1

Yes, this is what was needed!

Make sure the duplicate names are removed. Print the list of unique values from the `'type'` column:

[24]:

```
# viewing unique genre names
print(df['type'].unique())
```

```
['MOVIE' 'the movie' 'show' 'movies']
```

0.4.3 Conclusions

We detected three issues with the data:

- Incorrect header styles
- Missing values
- Duplicate rows and implicit duplicates

The headers have been cleaned up to make processing the table simpler.

All rows with missing values have been removed.

The absence of duplicates will make the results more precise and easier to understand.

Now we can move on to our analysis of the prepared data.

0.5 Stage 3. Data analysis

Based on the previous project stages, you can now define how the assumption will be checked. Calculate the average amount of votes for each score (this data is available in the `imdb_score` and `imdb_votes` columns), and then check how these averages relate to each other. If the averages for shows with the highest scores are bigger than those for shows with lower scores, the assumption appears to be true.

Based on this, complete the following steps:

- Filter the dataframe to only include shows released in 1999 or later.
- Group scores into buckets by rounding the values of the appropriate column (a set of 1-10 integers will help us make the outcome of our calculations more evident without damaging the quality of our research).
- Identify outliers among scores based on their number of votes, and exclude scores with few votes.
- Calculate the average votes for each score and check whether the assumption matches the results.

To filter the dataframe and only include shows released in 1999 or later, you will take two steps. First, keep only titles published in 1999 or later in our dataframe. Then, filter the table to only contain shows (movies will be removed).

```
[25]: # using conditional indexing modify df so it has only titles released after␣
      ↪1999 (with 1999 included)
      # give the slice of dataframe new name
      after_1999 = df[df['release_year'] >= 1999]
      print(after_1999)
```

```
                     name             character      role     title       type  \
1661          Jeff Probst        Himself - Host     ACTOR  Survivor       show
1952     Benicio del Toro  Franky Four Fingers     ACTOR    Snatch      MOVIE
1953        Dennis Farina            Cousin Avi     ACTOR    Snatch      MOVIE
1954         Vinnie Jones     Bullet Tooth Tony     ACTOR    Snatch      MOVIE
1955            Brad Pitt        Mickey O'Neil     ACTOR    Snatch      MOVIE
…                     …                     …   …      …        …
73854        A??da Morales               Maritza     ACTOR   Lokillo  the movie
73855     Adelaida Buscato             Mar??a Paz     ACTOR   Lokillo  the movie
73856  Luz Stella Luengas          Karen Bayona     ACTOR   Lokillo  the movie
73857          In??s Prieto                 Fanny     ACTOR   Lokillo  the movie
73858       Julian Gaviria               unknown  DIRECTOR   Lokillo  the movie

       release_year               genres  imdb_score  imdb_voted
1661           2000          ['reality']         7.4     24687.0
1952           2000  ['crime', 'comedy']         8.3    841435.0
1953           2000  ['crime', 'comedy']         8.3    841435.0
1954           2000  ['crime', 'comedy']         8.3    841435.0
1955           2000  ['crime', 'comedy']         8.3    841435.0
…                …                    …         …          …
73854          2021           ['comedy']         3.8        68.0
73855          2021           ['comedy']         3.8        68.0
73856          2021           ['comedy']         3.8        68.0
73857          2021           ['comedy']         3.8        68.0
73858          2021           ['comedy']         3.8        68.0

[69881 rows x 9 columns]
```

```
[26]: # repeat conditional indexing so df has only shows (movies are removed as␣
      ↪result)
      only_shows = after_1999[after_1999['type']== 'show']
      only_shows
```

```
[26]:                   name                 character      role      title  \
      1661          Jeff Probst            Himself - Host     ACTOR   Survivor
      2073        Mayumi Tanaka  Monkey D. Luffy (voice)     ACTOR  One Piece
      2074          Kazuya Nakai      Roronoa Zoro (voice)     ACTOR  One Piece
```

```
2075         Akemi Okamura             Nami (voice)     ACTOR  One Piece
2076      Kappei Yamaguchi            Usopp (voice)     ACTOR  One Piece
…                    …                      …       …        …
73831     Maneerat Kam-Uan                       Ae     ACTOR  Let's Eat
73832    Rudklao Amratisha                  unknown     ACTOR  Let's Eat
73833      Jaturong Mokjok                  unknown     ACTOR  Let's Eat
73834     Pisamai Wilaisak                  unknown     ACTOR  Let's Eat
73835  Sarawut Wichiensarn                  unknown  DIRECTOR  Let's Eat

       type  release_year                                             genres  \
1661   show          2000                                        ['reality']
2073   show          1999  ['animation', 'action', 'comedy', 'drama', 'fa…
2074   show          1999  ['animation', 'action', 'comedy', 'drama', 'fa…
2075   show          1999  ['animation', 'action', 'comedy', 'drama', 'fa…
2076   show          1999  ['animation', 'action', 'comedy', 'drama', 'fa…
…     …           …                                                  …
73831  show          2021                                ['drama', 'comedy']
73832  show          2021                                ['drama', 'comedy']
73833  show          2021                                ['drama', 'comedy']
73834  show          2021                                ['drama', 'comedy']
73835  show          2021                                ['drama', 'comedy']

       imdb_score  imdb_voted
1661          7.4     24687.0
2073          8.8    117129.0
2074          8.8    117129.0
2075          8.8    117129.0
2076          8.8    117129.0
…            …          …
73831         8.2         5.0
73832         8.2         5.0
73833         8.2         5.0
73834         8.2         5.0
73835         8.2         5.0

[13430 rows x 9 columns]
```

The scores that are to be grouped should be rounded. For instance, titles with scores like 7.8, 8.1, and 8.3 will all be placed in the same bucket with a score of 8.

[28]:
```
# rounding column with scores

only_shows['imdb_score'] = only_shows['imdb_score'].round()
#checking the outcome with tail()
only_shows.head()
```

```
/tmp/ipykernel_26/2130509769.py:3: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
```

```
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  only_shows['imdb_score'] = only_shows['imdb_score'].round()
```

[28]:

```
                   name                character   role      title  type  \
1661        Jeff Probst           Himself - Host  ACTOR   Survivor  show
2073      Mayumi Tanaka  Monkey D. Luffy (voice)  ACTOR  One Piece  show
2074       Kazuya Nakai      Roronoa Zoro (voice)  ACTOR  One Piece  show
2075      Akemi Okamura             Nami (voice)  ACTOR  One Piece  show
2076  Kappei Yamaguchi            Usopp (voice)  ACTOR  One Piece  show

      release_year                                             genres  \
1661          2000                                        ['reality']
2073          1999  ['animation', 'action', 'comedy', 'drama', 'fa…
2074          1999  ['animation', 'action', 'comedy', 'drama', 'fa…
2075          1999  ['animation', 'action', 'comedy', 'drama', 'fa…
2076          1999  ['animation', 'action', 'comedy', 'drama', 'fa…

      imdb_score  imdb_voted
1661         7.0     24687.0
2073         9.0    117129.0
2074         9.0    117129.0
2075         9.0    117129.0
2076         9.0    117129.0
```

<div class="alert alert-success"; style="border-left: 7px solid green"> Reviewer's comment, v. 1

All the transformations were performed absolutely correctly

It is now time to identify outliers based on the number of votes.

[29]:

```
# Use groupby() for scores and count all unique values in each group, print the␣
 ↪result
score_count = only_shows.groupby('imdb_score')['imdb_voted'].count()
score_count
```

[29]:

```
imdb_score
2.0       24
3.0       27
4.0      180
5.0      592
6.0     2494
7.0     4706
8.0     4842
9.0      557
10.0       8
Name: imdb_voted, dtype: int64
```

<div class="alert alert-success"; style="border-left: 7px solid green"> Reviewer's comment, v. 1

The dataframe was filtered and grouped flawlessly

Based on the aggregation performed, it is evident that scores 2 (24 voted shows), 3 (27 voted shows), and 10 (only 8 voted shows) are outliers. There isn't enough data for these scores for the average number of votes to be meaningful.

To obtain the mean numbers of votes for the selected scores (we identified a range of 4-9 as acceptable), use conditional filtering and grouping.

[30]:

```
# filter dataframe using two conditions (scores to be in the range 4-9)
filtered_dataframe = only_shows[(only_shows['imdb_score'] >= 4) &␣
 ↪(only_shows['imdb_score'] <= 9)]

# group scores and corresponding average number of votes, reset index and print␣
 ↪the result

grouped_scores = filtered_dataframe.groupby('imdb_score')['imdb_voted'].mean().
 ↪reset_index()
grouped_scores
```

[30]:

```
   imdb_score     imdb_voted
0         4.0    5277.583333
1         5.0    3143.942568
2         6.0    3481.717322
3         7.0    8727.068211
4         8.0   30299.460967
5         9.0  126904.109515
```

Now for the final step! Round the column with the averages, rename both columns, and print the dataframe in descending order.

[31]:

```
# round column with averages
grouped_scores['imdb_voted'] = grouped_scores['imdb_voted'].round()

# rename columns
rename_columns = grouped_scores.rename(columns={'imdb_score':'scores',␣
 ↪'imdb_voted':'votes'})

# print dataframe in descending order
descending_order = rename_columns.sort_values(by= 'votes', ascending = False)
descending_order
```

[31]:

```
   scores     votes
5     9.0  126904.0
4     8.0   30299.0
3     7.0    8727.0
0     4.0    5278.0
```

```
2    6.0    3482.0
1    5.0    3144.0
```

The assumption macthes the analysis: the shows with the top 3 scores have the most amounts of votes.

<div class="alert alert-success"; style="border-left: 7px solid green"> Reviewer's comment, v. 1

Great! Also correct rounding and grouping in this section

0.6 Conclusion

The research done confirms that highly-rated shows released during the "Golden Age" of television also have the most votes. While shows with score 4 have more votes than ones with scores 5 and 6, the top three (scores 7-9) have the largest number. The data studied represents around 94% of the original set, so we can be confident in our findings.

<div class="alert alert-success"; style="border-left: 7px solid green"> Reviewer's comment, v. 1

Overall conclusion is an important part, where we should include the summary of the outcomes of the project.