Data and Artificial Intelligence
Cyber Shujaa Program

Week 2 Assignment
Data Wrangling using Python onKaggle Notebook

Student Name: Violet Joy

Student ID: CS-DA01-25025

INTRODUCTION

Data wrangling, also known as data cleaning or data preprocessing, is a fundamental step in the data analysis pipeline. Before any meaningful analysis or modeling can be performed, raw data must be transformed into a clean and structured format. This report documents the data wrangling process carried out on a Netflix_shows dataset using python on Kaggle Notebook, an interactive environment that combines code, data visualization, and narrative text.

The primary objective of this assignment is to demonstrate proficiency in identifying and handling common data quality issues such as missing values, duplicates, inconsistent formatting, and outliers.

Data wrangling has the following key steps:

- Discovery
- Structuring
- Cleaning
- Enriching
- Validating
- Publishing

Tasks Completed

/kaggle/input/netflix-shows/netflix_titles.csv

DATA SCIENCE PROJECT: DATA WRANGLING

This project outlines the steps in data wrangling which include discovery, structuring, cleaning, enriching, validating and publishing as I showcase my work using python on netflix.

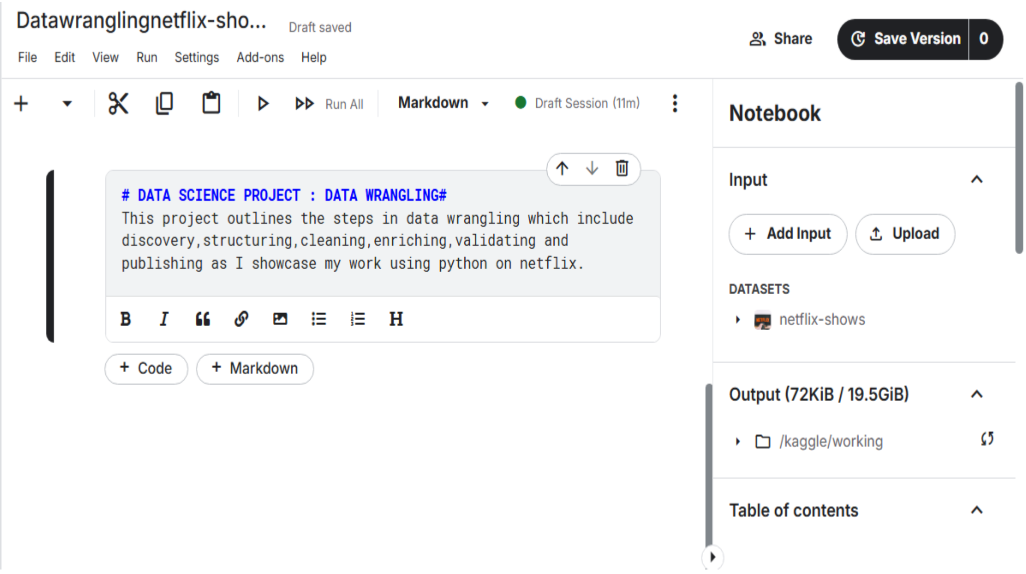

Fig 1; Introduction to my assignment

STEP 1: DISCOVERY

#import the data to a Pandas Dataframe

df=pd.read_csv('/kaggle/input/netflix-shows/netflix_titles.csv')

#quick overview of the dataset

df.info()

df.describe()

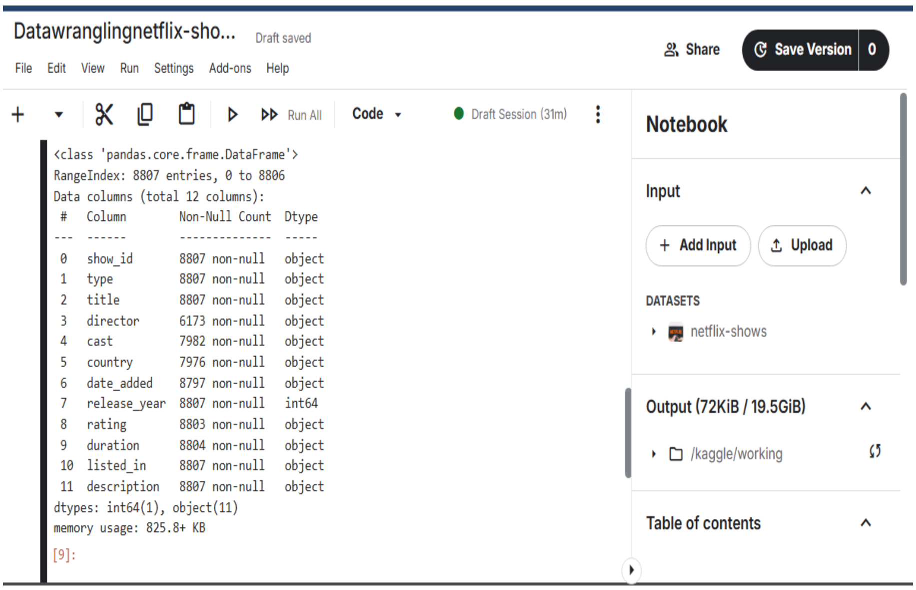

Fig 2; Output of understanding the dataset in columns and rows

#number of rows and columns

print("shape of dataset(R x C):",df.shape)

#list of all column names

print("columns in the dataset:\n",df.columns.tolist())

#group and count of missing values in each column

print("missing values per column :\n",df.isnull().sum())

#group and count of duplicate rows

print("number of duplicate rows :", df.duplicate().sum())

(DataFrame' object has no attribute 'duplicate' ;outcome)

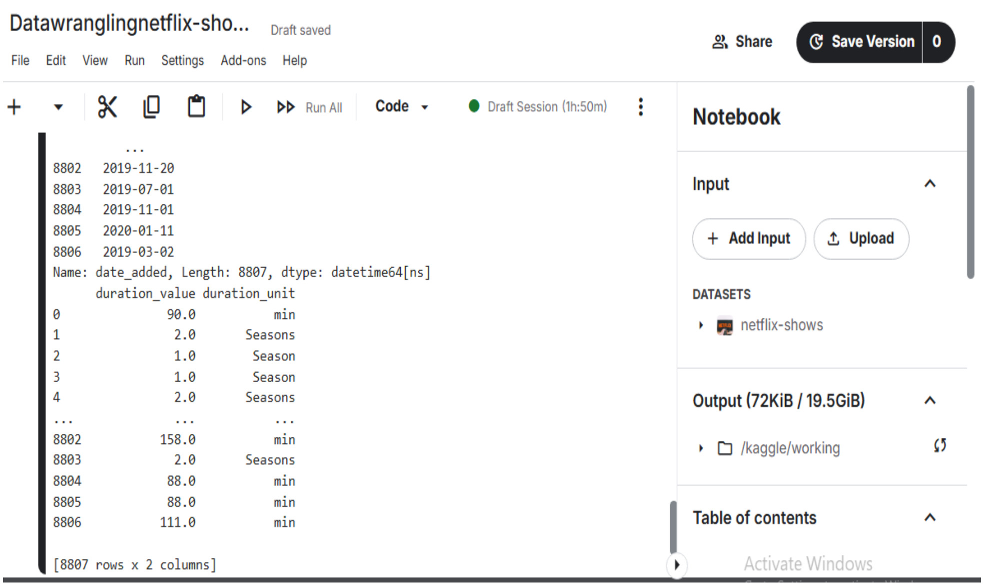

Fig 3; Output of structuring and formatting columns

STEP 2. STRUCTURING

#convert 'date_added'to datetime

df['date_added']=pd.to_datetime(df['date_added'],format='mixed')

print(df['date_added'])

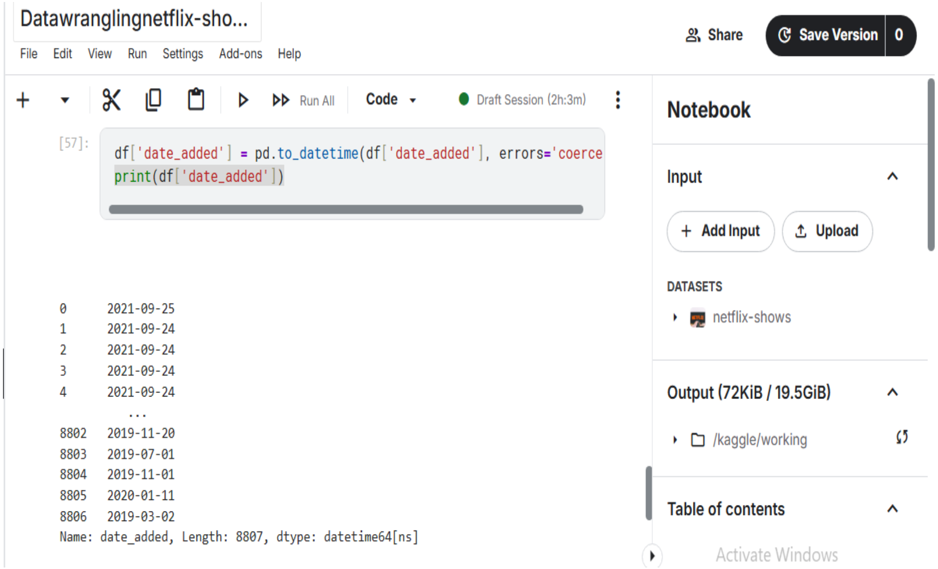

Fig 4; Output for structuring dates

#separate 'duration ‘into numeric value and unit

df[['duration_value', 'duration_unit']] = df['duration'].str.extract(r'(\d+)\s*(\w+)')

#convert 'duration_value'to numeric

df['duration_value']=pd.to_numeric(df['duration_value'])

#viewing resulting columns

-Another structuring way of data using corce function

print(df[['duration_value','duration_unit']])

df['date_added'] = pd.to_datetime(df['date_added'], errors='coerce

```
df = df.assign(
    duration_value=df['duration'].str.extract(r'(\d+)').astype(float),
    duration_unit=df['duration'].str.extract(r'\d+\s*(\w+)')
)
print(df[['duration_value','duration_unit']])
```

If you want to unify the duration into one unit (e.g., minutes):

```
def normalize_duration(val, unit):
    if unit == 'min':
        return val
    elif unit == 'h':
        return val * 60
```

```
    elif unit == 'Season':
        return val * 600  # just an example assumption
    return pd.NA
df['duration_minutes'] = df.apply(lambda row: normalize_duration(row['duration_value'], row['duration_unit']), axis=1)
```

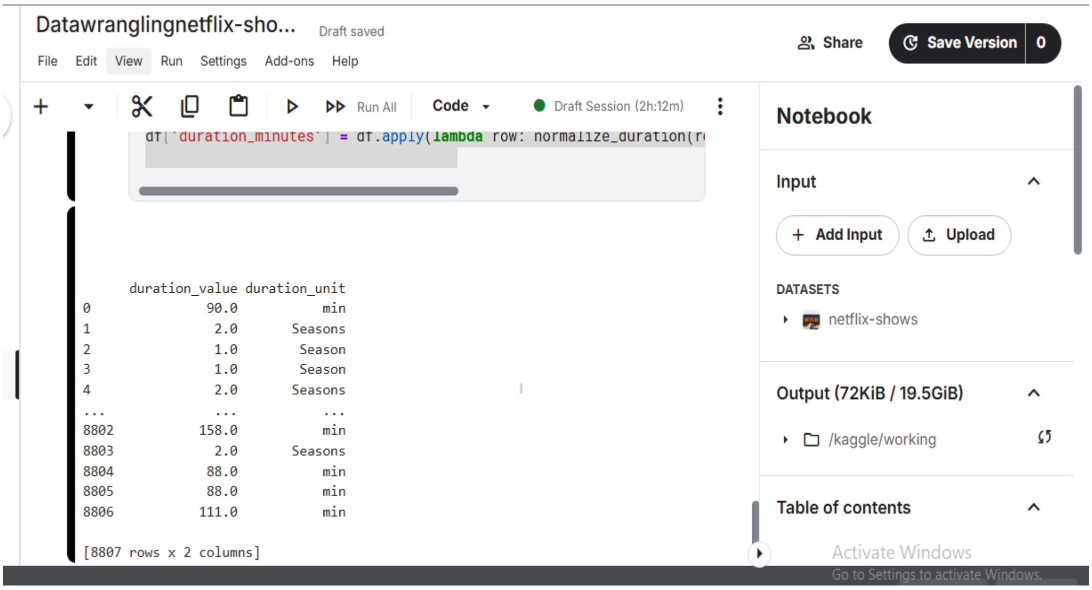

Fig 5; Output on normalizing dataset

STEP 3.CLEANING

#check for duplicate rows

print("duplicate rows before:",df.duplicated().sum())

 # Drop duplicate rows if any

df = df.drop_duplicates()

#remove irrelevant information that is description

df = df.drop(columns=['description'],inplace=True)

 # Impute Director values by using relationship between cast and director

 # List of Director-Cast pairs and the number of times they appear

df['dir_cast'] = df['director'].fillna('Unknown') + '---' + df['cast'].fillna('Unknown')

#counts unique values

counts = df['dir_cast'].value_counts()

#checks if repeated 3 or more

filtered_counts = counts[counts >= 3]

#gets the values

filtered_values = filtered_counts

#convert to list

lst_dir_cast = list(filtered_values)

dict_direcast = {}

for i in lst_dir_cast:

    if isinstance(i, str) and '---' in i:

        director, cast = i.split('---')

    dict_direcast[i] = ('director'.strip(), 'cast'.strip())

else:

        print(f"skipping non-string or malformed entry: {i}")

for i in range(len(dict_direcast)):

    df.loc[(df['director'].isna()) & (df['cast'] == list(dict_direcast.items())[i][1]),'director'] = list(dict_direcast.items())[i][0]

 # Assign Not Given to all other director fields

df.loc[df['director'].isna(),'director'] ='Not Given'

#confirm no missing values

df.isnull().sum()

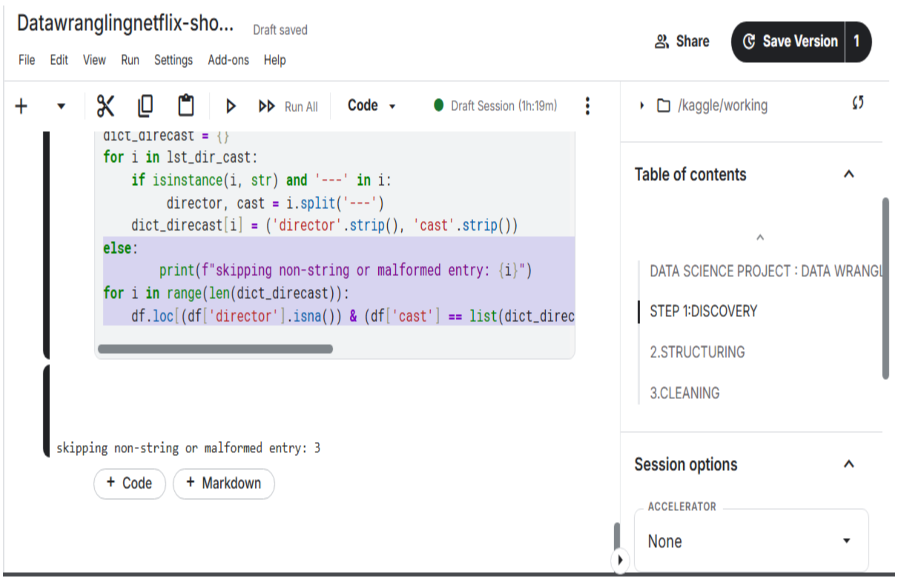

Fig 6;Output on missing value director

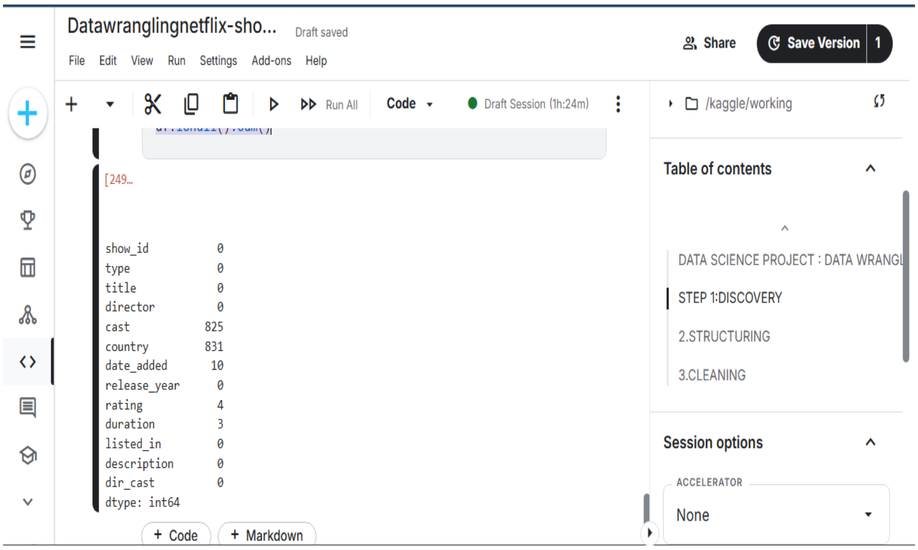

Fig 7;Output on assigning missing values

#Use directors to fill missing countries

directors = df['director']

countries = df['country']

#pair each director with their country use zip() to get an iterator of tuples

pairs = zip(directors, countries)

 # Convert the list of tuples into a dictionary

dir_cntry = dict(list(pairs))

#Use directors to fill missing countries

directors = df['director']

countries = df['country']

#pair each director with their country use zip() to get an iterator of tuples

pairs = zip(directors, countries)

 # Convert the list of tuples into a dictionary

dir_cntry = dict(list(pairs))

 # Assign Not Given to all other country fields

df.loc[df['country'].isna(),'country'] = 'Not Given'

 # Assign Not Given to all other fields

df.loc[df['cast'].isna(),'cast'] = 'Not Given'

#confirm no missing values

df.isnull().sum()

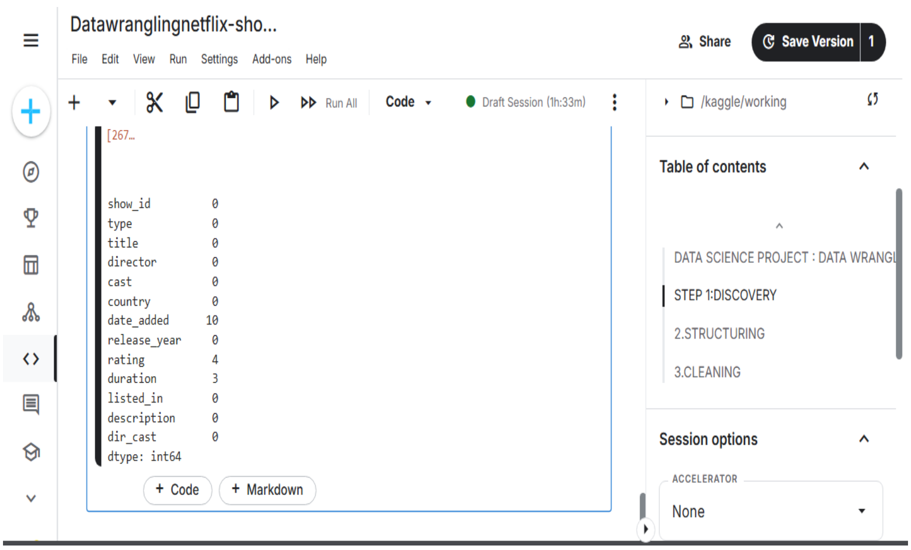

Fig 8;Output on structuring and formatting country

STEP 4. ENRICHING

 # check if there are any added_dates that come before release_year

import datetime as dt

sum(df['date_added'].dt.year < df['release_year'])

df.loc[(df['date_added'].dt.year < df['release_year']),['date_added','release_year']]

 # sample some of the records and check that they have been accurately replaced

df.iloc[[1551,1696,2920,3168]]

#Confirm that no more release_year inconsistencies

sum(df['date_added'].dt.year < df['release_year'])

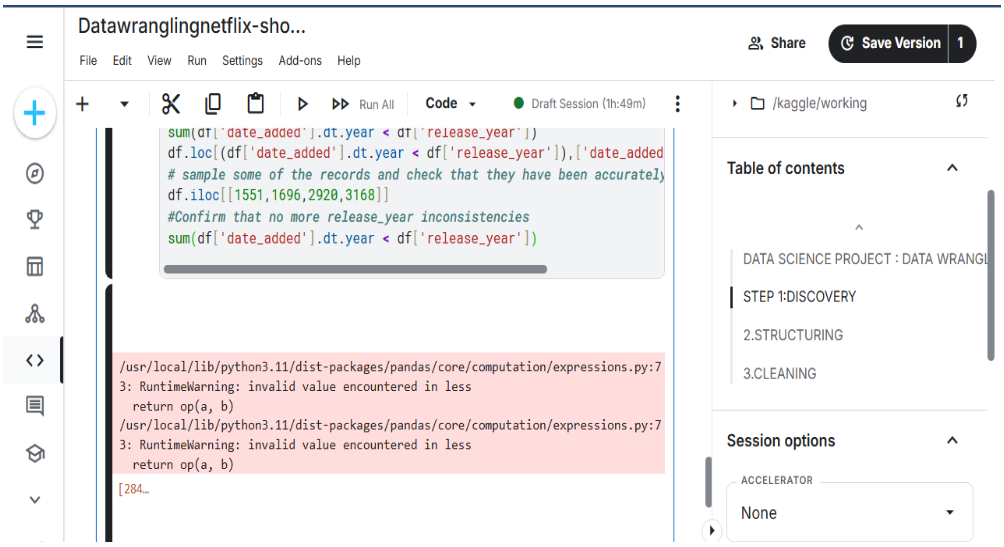

Fig 9; Output on checking on errors

Errors

 # Ensure 'date_added' is in datetime format

df['date_added'] = pd.to_datetime(df['date_added'], errors='coerce')

 # Identify inconsistencies

inconsistencies = df['date_added'].dt.year < df['release_year']

print("Number of inconsistencies:", inconsistencies.sum())

 # View problematic rows

print(df.loc[inconsistencies, ['date_added', 'release_year']])

Fixing inconsistencies

 # Ensure 'date_added' is in datetime format

df['date_added'] = pd.to_datetime(df['date_added'], errors='coerce')

 # Identify inconsistent rows

inconsistencies = df['date_added'].dt.year < df['release_year']

#Report how many rows will be fixed

print("Number of inconsistencies to fix:", inconsistencies.sum())

#Fix: Set release_year to match the year of date_added

df.loc[inconsistencies, 'release_year'] = df.loc[inconsistencies, 'date_added'].dt.year

 # Confirm fix

print("Remaining inconsistencies after fix:", (df['date_added'].dt.year < df['release_year']).sum())

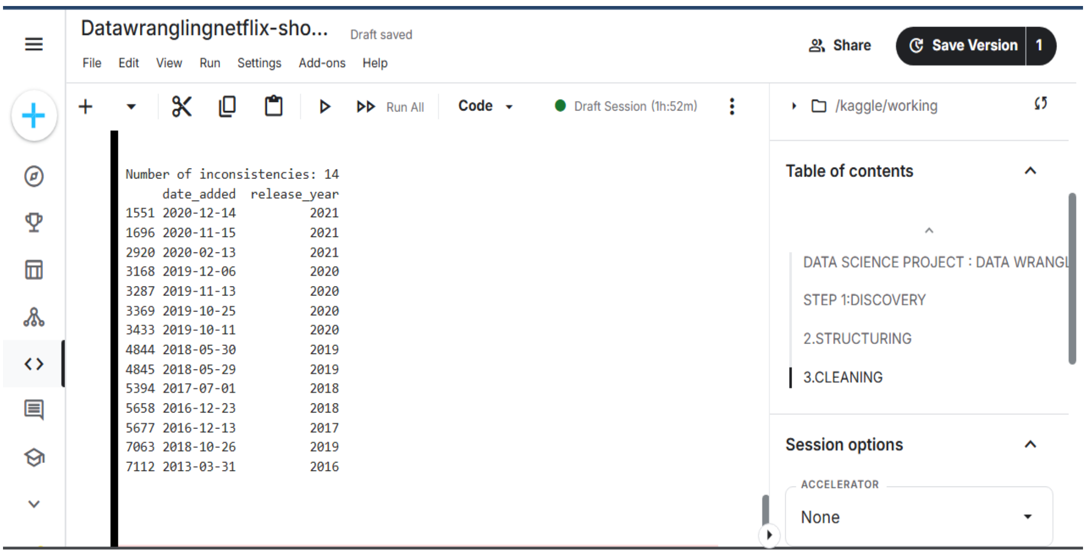

Fig 10; Output on inconsistencies

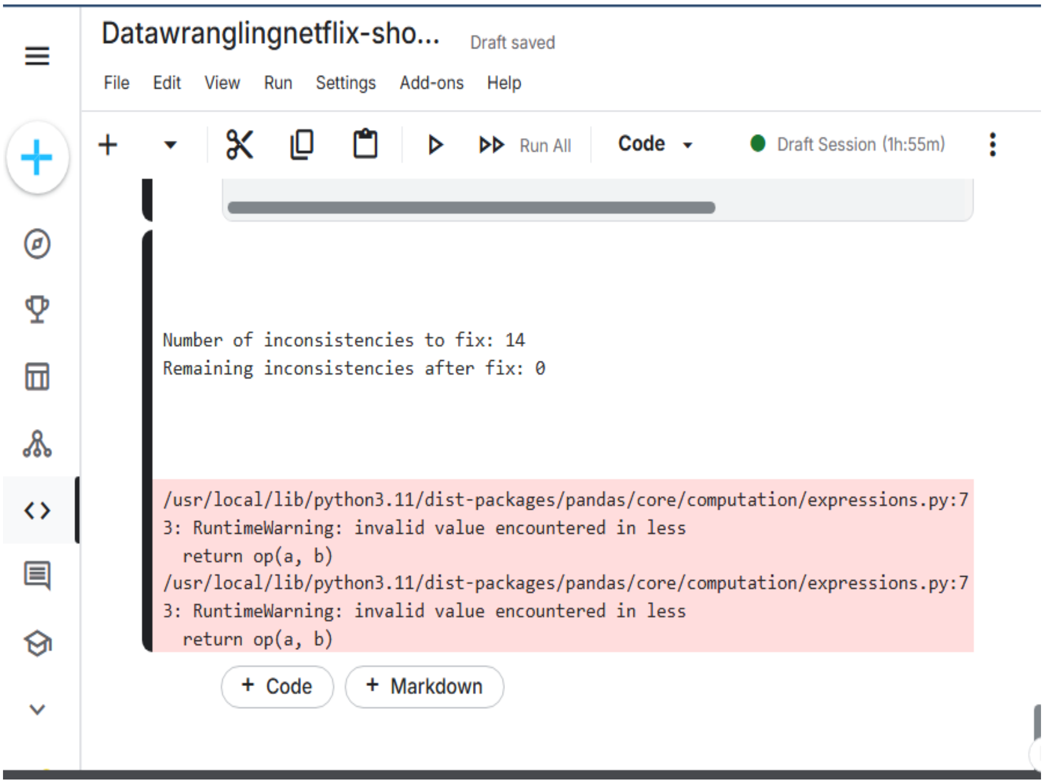

Fig 11;Output on fixing inconsistencies

STEP 5. VALIDATING

```
# Ensure 'df' exists
try:
    df
except NameError:
    print("Error: DataFrame 'df' is not defined.")
else:
    # Convert 'date_added' to datetime if column exists
    if 'date_added' in df.columns:
        df['date_added'] = pd.to_datetime(df['date_added'], errors='coerce')
    else:
        print("Warning: 'date_added' column not found in DataFrame.")
    # Convert 'duration_value' to numeric if column exists
    if 'duration_value' in df.columns:
        df['duration_value'] = pd.to_numeric(df['duration_value'], errors='coerce')
    else:
print("Warning: 'duration_value' column not found in DataFrame.")
```

#check resulting data types

print("\nColumn data types:")

print(df.dtypes)

#check for nulls created during coercion

print("\nMissing values after conversion:")

cols = ['date_added', 'duration_value']

existing_cols = [col for col in cols if col in df.columns]

print(df[existing_cols].isna().sum())

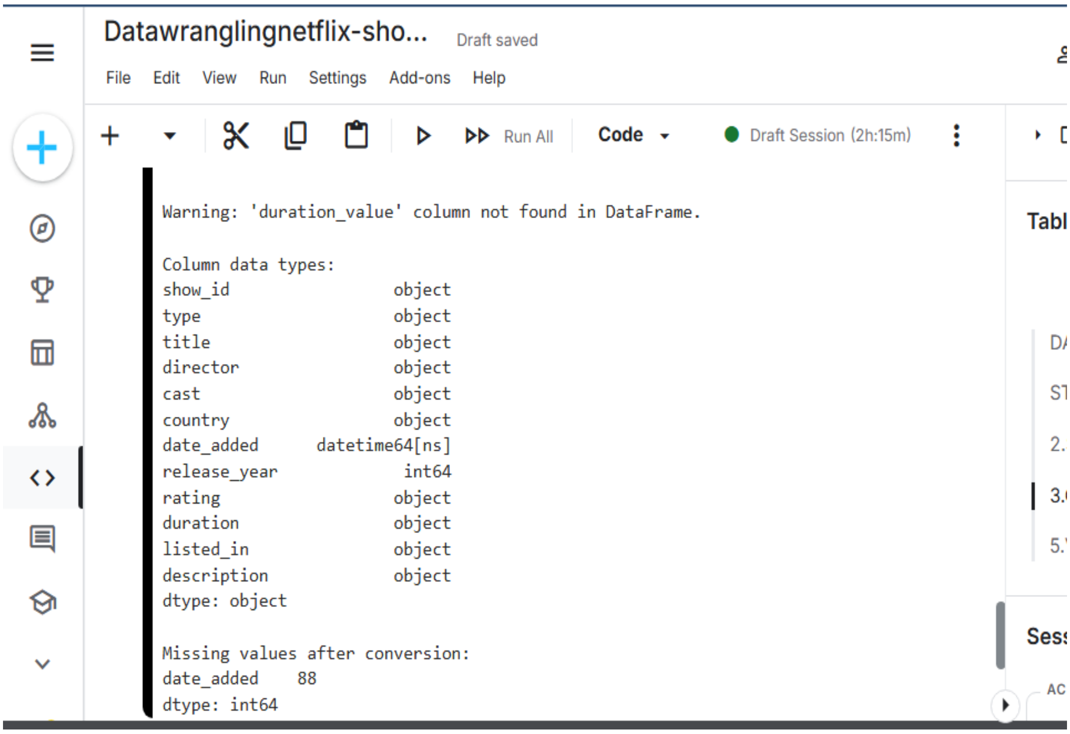

Fig 12; Output on validating

#sampling a row to check visually

df.sample(10)

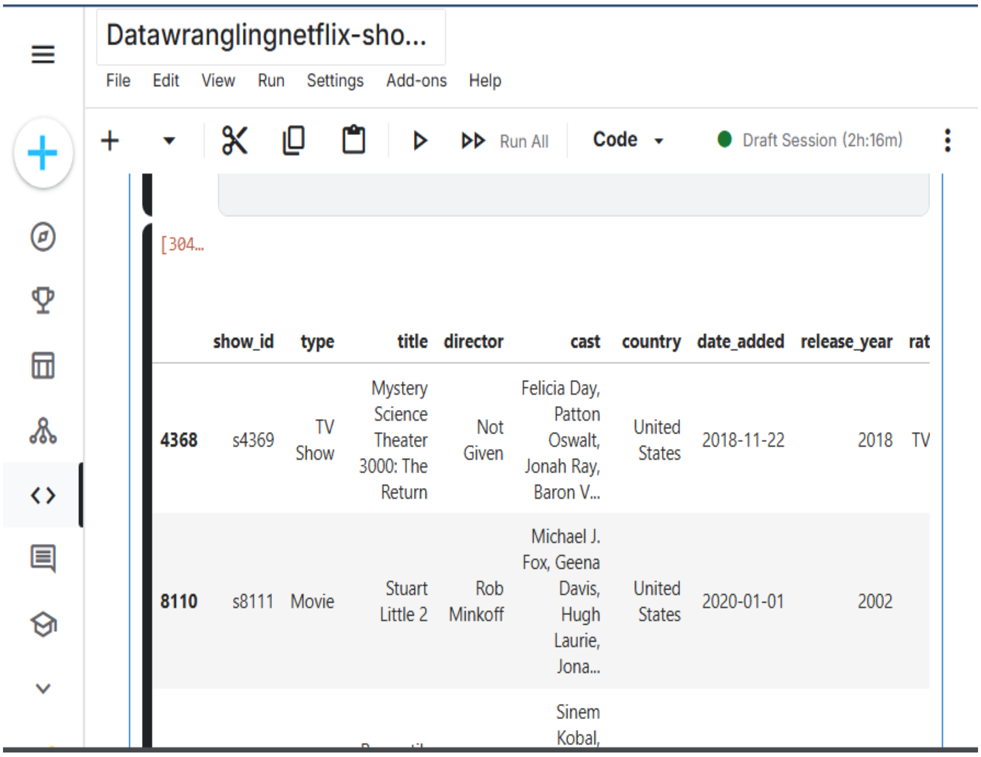

Fig 13; Output on validating sample(10)

#reseting the index

df_reset = df.reset_index(drop=True)

STEP 6.PUBLISHING

 # Save as CSV

df.to_csv('/kaggle/working/cleaned_netflix.csv', index=False)

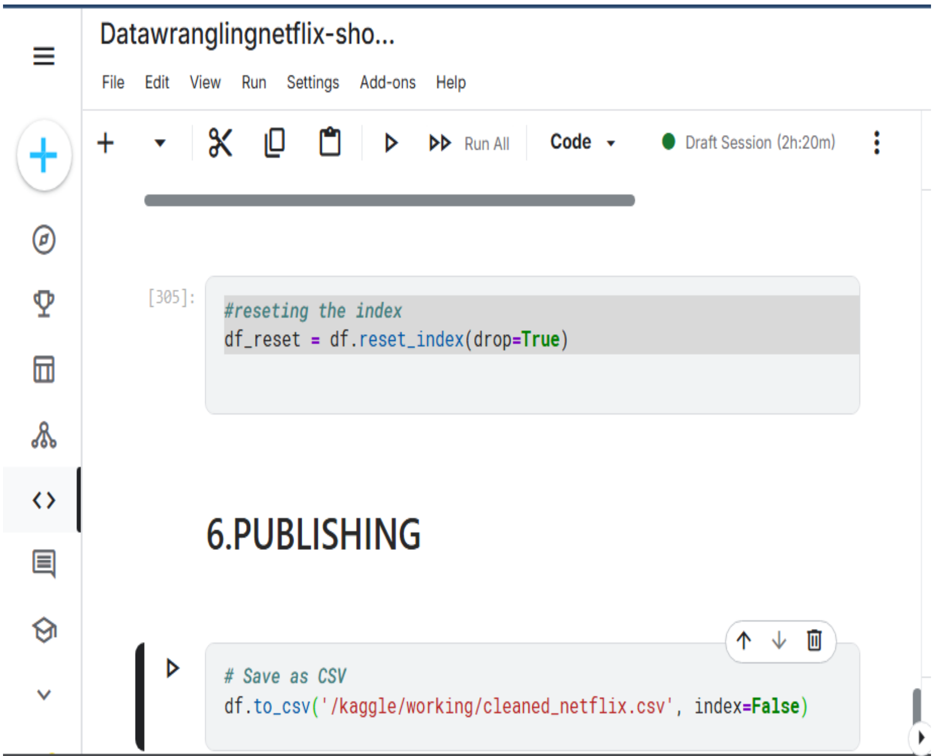

Fig 14; Resetting and publishing

Link to Code:

https://www.kaggle.com/code/joyviolet/datawranglingnetflix-shows

CONCLUSION

In this report, I have documented the complete data wrangling process I performed using Python in a Kaggle Notebook environment. The primary goal was to clean and prepare the dataset for further analysis by addressing common data quality issues. This involved inspecting the dataset, identifying and handling missing values, removing duplicates, correcting inconsistent data types, renaming columns, and formatting data to ensure consistency.

Python libraries such as **Pandas** and **NumPy** proved to be powerful tools for performing these tasks efficiently. With their robust functionalities, I was able to carry out various transformations and validate each step with clear output and visual confirmation. The use of built-in methods allowed for the detection of anomalies and streamlined the overall cleaning process.

Through this systematic approach, the raw dataset was successfully transformed into a clean and structured format that is ready for deeper exploration, visualization, or machine learning modeling. This process not only improves the accuracy and reliability of any future analysis but also underscores the importance of data wrangling as a foundational step in any data science or analytics workflow.

Overall, this exercise demonstrates how thoughtful data preprocessing is essential for extracting meaningful insights from data and building high-quality, data-driven solutions and it has been really impactful.