UNIT 5_II

GitHub

PREPROCESSING | DATA CLEANING → DATA INCONSISTENCIES/ ANOMALIES

1. Data Inconsistency

Data inconsistencies are errors or contradictions in a dataset, where the same data is represented in different, conflicting, mismatched, or illogical ways across a dataset. In machine learning, inconsistent data can **mislead the model**, reduce accuracy, and increase preprocessing complexity.

In **data cleaning**, handling data inconsistencies means identifying and correcting these mismatches so the data becomes **accurate, uniform, and reliable** for training machine learning models.

1.1. Data Inconsistency by Data Type / Format

```python
# Import necessary libraries
import pandas as pd
import numpy as np
from datetime import datetime
import re

# Load the dataset from GitHub
df =
pd.read_csv("https://raw.githubusercontent.com/tabassumgulfaraz-
ds/machine_learning_1.0/main/files_and_datasets/f_ds5_II/inconsisten
t_data.csv")

# Check the shape i.e., number of rows and columns also called
dimensions of the dataset
print(f"Dataset Shape: {df.shape}")

# Display the first few rows of the dataset to understand its structure
and identify any inconsistencies
df.head()
```

Output:

```
Dataset Shape: (30, 11)
```

	Name	Age	Gender	Country	DateOfBirth	IsActive	Price	Salary	RegistrationDate	FatherAge	SonAge
0	James Taylor	25	M	USA	1/15/1999	TRUE	$100	50000	1/10/2020	55	25
1	james taylor	300	Male	U.S.A	15/01/1999	1	100	NaN	10/1/2020	50	25
2	Sarah Connor	28	F	United State	5/20/1996	Yes	USD 100	60000	6/15/2019	58	28
3	Sarah Connor	28	female	United State of America	20-05-1996	TRUE	$120	60000	15-06-2019	58	28
4	Michael Scott	150	m	USA	3/10/2010	FALSE	85.5	0	3/1/2021	30	25

1.1.1. Numerical Data

- **Example:** Age recorded as 25 (year) in one row and 250 (months) in another for the same person. If Age is greater than 100 consider as months.
- **Issues:** Outliers, wrong units, conflicting measurements.
- **Cleaning Approach:** Apply validation rules, detect outliers, cross-check with reference ranges.

```
# Identify unique values in the 'Age' column to find inconsistencies
sorted(df['Age'].unique())
```

Output:

```
[np.int64(13),
 np.int64(15),
 np.int64(20),
 np.int64(22),
 np.int64(25),
 np.int64(27),
 np.int64(28),
 np.int64(29),
 np.int64(32),
 np.int64(35),
 np.int64(38),
 np.int64(42),
 np.int64(45),
 np.int64(50),
 np.int64(150),
 np.int64(156),
 np.int64(180),
 np.int64(240),
 np.int64(264),
 np.int64(300),
 np.int64(324),
 np.int64(420)]
```

```
# Identify ages > 100 (these are in months, convert to years)
df['Age'] = df['Age'].apply(lambda x: x // 12 if x > 100 else x)
```

```
# Display summary statistics of the 'Age' column to check for any remaining inconsistencies
df['Age'].describe()
```

Output:

```
count    30.000000
mean     29.633333
std      10.990539
min      12.000000
25%      22.000000
50%      28.500000
75%      37.250000
max      50.000000
Name: Age, dtype: float64
```

```
# Display unique values in the 'Age' column after cleaning to confirm that all ages are now in years
sorted(df['Age'].unique())
```

Output:

```
[np.int64(12),
 np.int64(13),
 np.int64(15),
 np.int64(20),
 np.int64(22),
 np.int64(25),
 np.int64(27),
 np.int64(28),
 np.int64(29),
 np.int64(32),
 np.int64(35),
 np.int64(38),
 np.int64(42),
 np.int64(45),
 np.int64(50)]
```

```
# Display the first few rows of the cleaned dataset to verify the changes
df['Age'].head()
```

Output:

```
0    25
1    25
2    28
3    28
4    12
Name: Age, dtype: int64
```

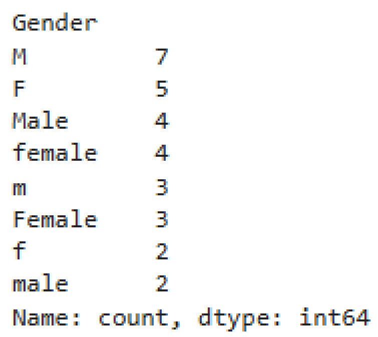
1.1.2. Categorical / Text Data

- **Example:** Gender recorded as M, Male, male, or m in different rows.

- **Issues:** Different spellings, synonyms, inconsistent abbreviations.
- **Cleaning Approach:** Standardization, mapping to a canonical form, case normalization.

```
# Display unique values in the 'Gender' column to identify inconsistencies
df['Gender'].value_counts()
```

Output:



```
Gender
M         7
F         5
Male      4
female    4
m         3
Female    3
f         2
male      2
Name: count, dtype: int64
```

```
# Standardize Gender values
gender_mapping = {
    'M': 'male',
    'male': 'male',
    'm': 'male',
    'Male': 'male',
    'F': 'female',
    'female': 'female',
    'f': 'female',
    'Female': 'female'
}

# Apply the mapping to standardize the 'Gender' column
df['Gender'] = df['Gender'].map(gender_mapping)

# Display the unique values in the 'Gender' column after cleaning to confirm that all values are standardized
df['Gender'].value_counts()
```

Output:

```
Gender
male      16
female    14
Name: count, dtype: int64
```

1.1.3. Date / Time Data

- **Example:** 01/02/2025 in one row, 2025-02-01 in another (DD/MM/YYYY vs YYYY-MM-DD).
- **Issues:** Different formats, wrong time zones, impossible dates (like 30 Feb).
- **Cleaning Approach:** Convert to a unified format (ISO standard), detect invalid dates.

```python
df['DateOfBirth'].head(10)
```

Output:

```
0     1/15/1999
1    15/01/1999
2     5/20/1996
3    20-05-1996
4     3/10/2010
5     3/10/1989
6     8/25/1998
7    25/08/2004
8    11/30/1979
9    30/11/1979
Name: DateOfBirth, dtype: str
```

```python
# Function to parse different date formats
def standardize_date(date_str):
    if pd.isna(date_str):
        return np.nan
    try:
        # Try different date formats
        for fmt in ['%Y-%m-%d', '%d/%m/%Y', '%d-%m-%Y',
                    '%m/%d/%Y', '%m-%d-%Y']:
            try:
                return pd.to_datetime(date_str,
                          format=fmt).strftime('%Y-%m-%d')
            except:
                continue
        return np.nan
    except:
        return np.nan
```

```
df['DateOfBirth'] = df['DateOfBirth'].apply(standardize_date)

# After cleaning (ISO format YYYY-MM-DD):
df['DateOfBirth'].head(10)
```

Output:

```
0    1999-01-15
1    1999-01-15
2    1996-05-20
3    1996-05-20
4    2010-10-03
5    1989-10-03
6    1998-08-25
7    2004-08-25
8    1979-11-30
9    1979-11-30
Name: DateOfBirth, dtype: str
```

1.1.4. Boolean / Binary Data

- **Example:** True/False vs 1/0 vs Yes/No in the same column.
- **Issues:** Mixed representations can confuse algorithms.
- **Cleaning Approach:** Map all values to a single consistent representation.

```
# Check the unique values in the 'IsActive' column to identify inconsistencies
df['IsActive'].value_counts()
```

Output:

```
IsActive
TRUE     10
0         5
1         4
Yes       4
FALSE     3
No        2
yes       2
Name: count, dtype: int64
```

```
# Standardize Boolean values in 'IsActive' column to handle inconsistencies.
def standardize_boolean(value):
    if isinstance(value, str):
        value = value.strip().lower()
        if re.match(r'^(true|yes|1)$', value):
            return True
        elif re.match(r'^(false|no|0)$', value):
            return False
    elif isinstance(value, (int, float)):
        if value == 1:
            return True
        elif value == 0:
            return False
    return np.nan

# Apply the standardization function to the 'IsActive' column and
convert to lowercase string for consistency
df['IsActive'] =
  df['IsActive'].map(standardize_boolean).astype(str).str.lower()

# Display the unique values in the 'IsActive' column after cleaning
to confirm that all values are standardized
df['IsActive'].value_counts()
```

Output:

```
IsActive
true     20
false    10
Name: count, dtype: int64
```

1.1.5. Multi-format / Mixed-type Columns

- **Example:** Column Price having values "$100", 100, USD 100.
- **Issues:** Numeric algorithms can’t directly process strings.
- **Cleaning Approach:** Strip non-numeric symbols, convert to float or integer.

```
# Display the first few rows of the 'Price' column to identify
inconsistencies
df['Price'].head(10)
```

Output:

```
0      $100
1        100
2    USD 100
3      $120
4       85.5
5       85.5
6      $200
7        200
8        150
9      $150
Name: Price, dtype: str
```

```python
# Check the data type of the 'Price' column to confirm that it is not numeric due to inconsistencies
df['Price'].dtype
```

Output:

```
<StringDtype(storage='python', na_value=nan)>
```

```python
# Function to clean price values
def clean_price(price):
    try:
        # Remove currency symbols and text
        price_str = str(price)
        # Remove $, USD, commas, quotes, and spaces
        cleaned = re.sub(r'[\$,USD"\s]', '', price_str)
        return float(cleaned)
    except:
        return np.nan

# Apply the cleaning function to the 'Price' column and convert it into numeric
df['Price'] = df['Price'].apply(clean_price)

# Display the first few rows of the 'Price' column after cleaning to confirm that all values are now numeric and cleaned
df['Price'].head(10)
```

Output:

```
0    100.0
1    100.0
2    100.0
3    120.0
4     85.5
5     85.5
6    200.0
7    200.0
8    150.0
9    150.0
Name: Price, dtype: float64
```

```
# Check the data type of the 'Price' column to confirm that it is not numeric due to inconsistencies
df['Price'].dtype
```

Output:

```
dtype('float64')
```

1.1.6. Missing / Null Values

- **Example:** One system records missing salary as NaN, another as 0, and another as Unknown.
- **Issues:** Can create misinterpretation during analysis.
- **Cleaning Approach:** Replace with a consistent placeholder, impute, or remove rows.

```
# check the count of missing values in the 'Salary' column, isna() considers NaN, None, and pandas.Nat values as missing.
df['Salary'].isna().sum()
# Values like 0, unknown, or the string 'nan' are not considered missing by isna().sum().
```

Output:

```
np.int64(3)
```

```
# Display unique values in the 'Salary' column to identify inconsistencies
df['Salary'].unique()
```

Output:

```
<StringArray>
[  '50000',       nan,   '60000',       '0',   '45000', 'Unknown',   '75000',
   '55000',   '48000',   '62000',   '70000',   '58000',   '52000',   '65000',
   '47000',   '80000',   '59000',   '72000']
Length: 18, dtype: str
```

```
# Replace inconsistent missing values with NaN, standardize missing
values to NaN
df['Salary'] = df['Salary'].replace(['Unknown', 'NaN', 0, '0'],
                np.nan)

# Check the count of missing values in the 'Salary' column
df['Salary'].isna().sum()
```

Output:

```
np.int64(8)
```

```
# Convert Salary column to numeric type
df['Salary'] = pd.to_numeric(df['Salary'], errors='coerce')

# Option 1: Fill with median
# df['Salary'].fillna(df['Salary'].median(), inplace=True)
df['Salary'] = df['Salary'].fillna(df['Salary'].median())

# Option 2: Fill with mean
# df['Salary'].fillna(df['Salary'].mean(), inplace=True)
# df['Salary'] = df['Salary'].fillna(df['Salary'].mean())

df['Salary'].isna().sum()
```

Output:

```
np.int64(0)
```

1.1.7. Duplicated / Redundant Records

- **Example:** Two entries for the same customer with slight differences in spelling or address.
- **Issues:** Biases statistical distributions, misleads ML models.
- **Cleaning Approach:** Deduplication using fuzzy matching, key-based merging.

```
# Check the length of the dataset before removing duplicates
len(df)
```

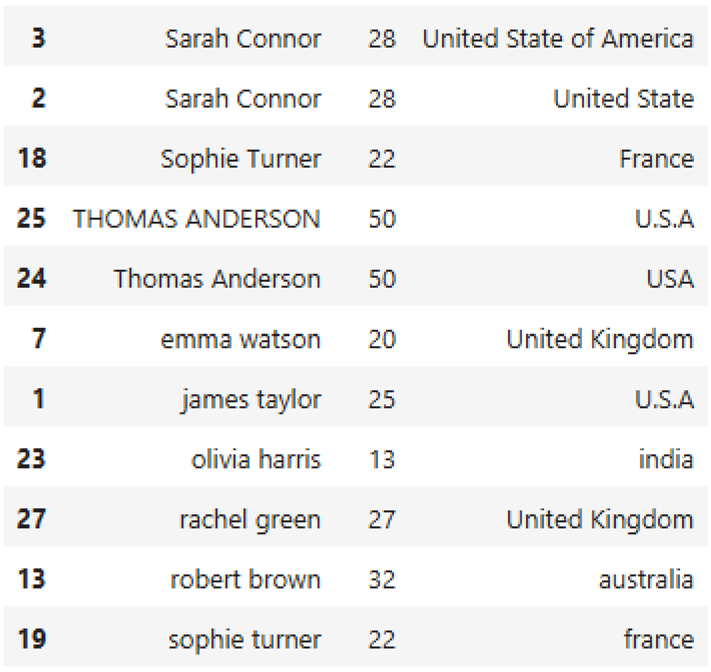
Output:

```
30
```

```
# Standardize Name column for duplicate detection
df['Name_Standardized'] = df['Name'].str.lower().str.strip()

# Display potential duplicates based on standardized names
df[df.duplicated(subset=['Name_Standardized'],
keep=False)][['Name', 'Age', 'Country']].sort_values('Name')
```

Output:

	Name	Age	Country
14	Anna Lee	35	China
15	Anna Lee	35	china
21	Chris Evans	38	United State
20	Chris Evans	38	USA
17	DAVID MILLER	29	GERMANY
16	David Miller	29	Germany
6	Emma Watson	20	UK
9	JOHN DOE	45	canada
0	James Taylor	25	USA
8	John Doe	45	Canada
10	Lisa Simpson	15	USA
11	Lisa Simpson	15	U.S.A
29	Mark Johnson	42	CANADA
28	Mark Johnson	42	Canada
5	Michael Scott	35	USA
4	Michael Scott	12	USA
22	Olivia Harris	13	India
26	Rachel Green	27	UK
12	Robert Brown	32	Australia



3	Sarah Connor	28	United State of America
2	Sarah Connor	28	United State
18	Sophie Turner	22	France
25	THOMAS ANDERSON	50	U.S.A
24	Thomas Anderson	50	USA
7	emma watson	20	United Kingdom
1	james taylor	25	U.S.A
23	olivia harris	13	india
27	rachel green	27	United Kingdom
13	robert brown	32	australia
19	sophie turner	22	france

```
# Remove duplicates keeping first occurrence
df = df.drop_duplicates(subset=['Name_Standardized'],
keep='first')


# Check the length of the dataset after removing duplicates
len(df)
```

Output:

```
15
```

```
# Drop the helper column
df.drop('Name_Standardized', axis=1, inplace=True)

# Update df (optional step to ensure df is updated, if we changed
df_cleaned to df)
df = df.copy()

# Check for any remaining duplicates in the dataset after cleaning.
df.duplicated().sum()
```

Output:

```
np.int64(0)
```

1.1.8. Logical Conflicts

- **Example:** A person's Date of Birth is 2010-05-10 but Age column shows 30.
- **Issues:** Internal inconsistencies can break model assumptions.
- **Cleaning Approach:** Recalculate dependent fields or flag errors for manual review.

```
# Logical Conflicts (Age vs DateOfBirth, FatherAge vs SonAge)
# Check 1: Age vs DateOfBirth consistency
current_year = 2026
def calculate_age_from_dob(dob):
    try:
        dob_date = pd.to_datetime(dob)
        return current_year - dob_date.year
    except:
        return np.nan

# Print function calculate_age_from_dob for testing
print(calculate_age_from_dob('1999-03-03'))  # Should return 27
```

Output:

```
27
```

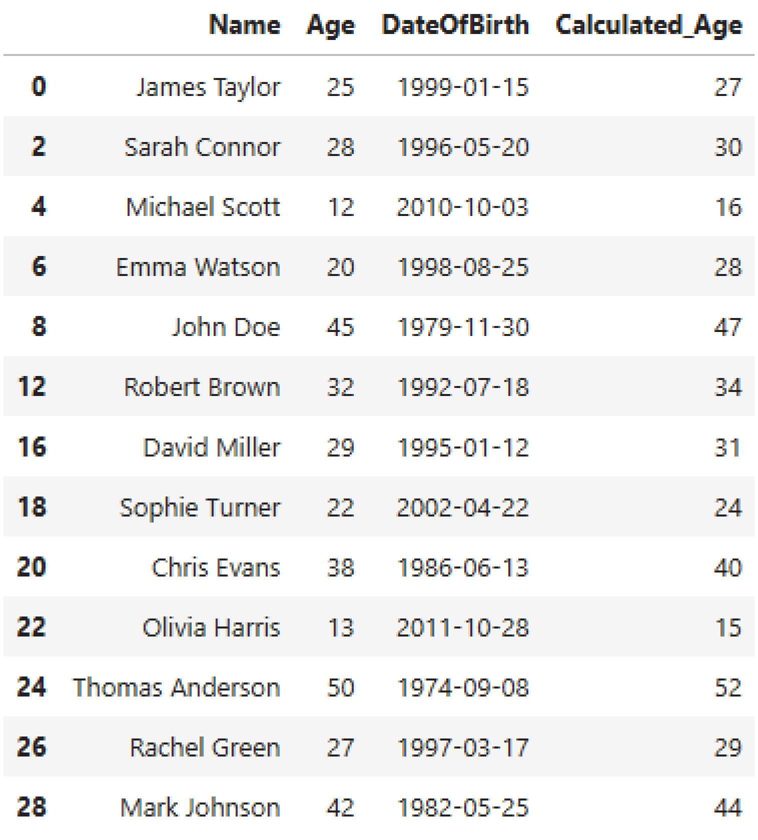
```
# Calculate age from DateOfBirth and create a new column
'Calculated_Age' to compare with the existing 'Age' column
df['Calculated_Age'] =
df['DateOfBirth'].apply(calculate_age_from_dob)

# Allowing a 1-year tolerance for minor discrepancies between the
reported age and the calculated age from DateOfBirth, we can
identify logical conflicts where the difference is greater than 1
year.
df['Age_Mismatch'] = abs(df['Age'] - df['Calculated_Age']) > 1

# This will give us the count of logical conflicts between Age and
DateOfBirth.
df['Age_Mismatch'].sum()
```

Output:

```
np.int64(13)
```

```
# Display rows with logical conflicts between Age and DateOfBirth
to investigate the discrepancies
df[df['Age_Mismatch']][['Name', 'Age', 'DateOfBirth',
'Calculated_Age']]
```

Output:



	Name	Age	DateOfBirth	Calculated_Age
0	James Taylor	25	1999-01-15	27
2	Sarah Connor	28	1996-05-20	30
4	Michael Scott	12	2010-10-03	16
6	Emma Watson	20	1998-08-25	28
8	John Doe	45	1979-11-30	47
12	Robert Brown	32	1992-07-18	34
16	David Miller	29	1995-01-12	31
18	Sophie Turner	22	2002-04-22	24
20	Chris Evans	38	1986-06-13	40
22	Olivia Harris	13	2011-10-28	15
24	Thomas Anderson	50	1974-09-08	52
26	Rachel Green	27	1997-03-17	29
28	Mark Johnson	42	1982-05-25	44

```
# Fix by recalculating from DateOfBirth
df['Age'] = df['Calculated_Age']
df.drop(['Calculated_Age', 'Age_Mismatch'], axis=1, inplace=True)

# Check 2: FatherAge vs SonAge (Father must be older)
df['Age_Conflict'] = df['SonAge'] >= df['FatherAge']

# This will give us the count of logical conflicts where SonAge
is greater than or equal to FatherAge.
df['Age_Conflict'].sum()
```

Output:

```
np.int64(0)
```

```
# Display rows with logical conflicts between FatherAge and SonAge
to investigate the discrepancies
df[df['Age_Conflict']][['Name', 'FatherAge', 'SonAge']]
```

Output:

```
   Name   FatherAge   SonAge
```

```
# Remove conflicting rows and keep only consistent data for analysis
df = df[~df['Age_Conflict']]

# Drop the helper column after removing conflicting rows
df.drop('Age_Conflict', axis=1, inplace=True)
```

1.1.9. Naming Conversion

- U.S.A, USA, United State, United State of America

```
# Install Libraries
%conda install pycountry thefuzz
# pip install pycountry thefuzz

# Import libraries for fuzzy matching
import pycountry
from thefuzz import process

# Display unique values in the 'Country' column to identify
inconsistencies
df['Country'].unique()
```

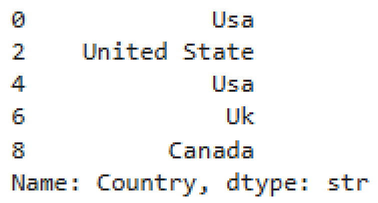
Output:

```
<StringArray>
[         'USA', 'United State',           'UK',       'Canada',
     'Australia',        'China',      'Germany',       'France',
        'India']
Length: 9, dtype: str
```

```
# .strip() → removes extra spaces from beginning and end
# .title() converts the first letter of each word to uppercase and
the remaining letters to lowercase.
# Automatic standardization using title case and strip
df['Country'] = df['Country'].str.strip().str.title()
```

```python
# Display the first few rows of the 'Country' column after cleaning
to confirm that all values are standardized
df['Country'].head()
```

Output:

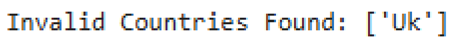


```
0              Usa
2     United State
4              Usa
6               Uk
8           Canada
Name: Country, dtype: str
```

```python
# Get official ISO country list/names
official_countries = [country.name for country in
pycountry.countries]

# ISO + Fuzzy Matching for Country Standardization
def clean_country(name):

    # 1 Try direct ISO lookup first (safest)
    try:
        return pycountry.countries.lookup(name).name
    except:
        pass

    # 2 Use fuzzy matching for longer names only (avoid short
      confusion like uk)
    if len(name) > 4:
        match, score = process.extractOne(name,
                          official_countries)
        if score >= 90:
            return match

    # 3 If not matched, return original (we will inspect later)
    return name

# Apply the cleaning function to the 'Country' column
```

```
df['Country'] = df['Country'].apply(clean_country)

# List to store invalid countries that are not matched to ISO
standards
invalid = []
for country in df['Country'].unique():
    try:
        pycountry.countries.lookup(country)
    except:
        invalid.append(country)

print("Invalid Countries Found:", invalid)
```

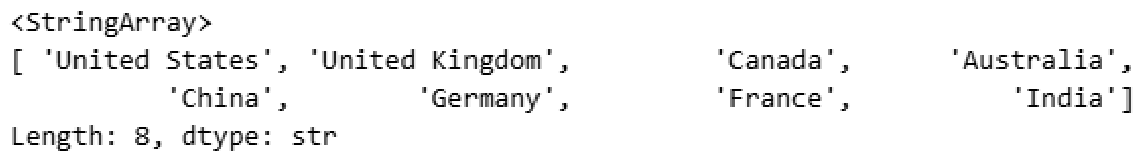
Output:



```
Invalid Countries Found: ['Uk']
```

```
# Manually fix remaining invalid countries based on inspection
manual_fix = {
    'uk': 'United Kingdom',
    'u.k': 'United Kingdom',
    'Uk': 'United Kingdom',
    'U.K': 'United Kingdom'
}

df['Country'] = df['Country'].replace(manual_fix)

# Display unique values in the 'Country' column to identify
inconsistencies
df['Country'].unique()
```

Output:



```
<StringArray>
[ 'United States', 'United Kingdom',          'Canada',      'Australia',
          'China',         'Germany',          'France',          'India']
Length: 8, dtype: str
```

1.1.10. Typographical Mistake

- James Taylor, james taylor

```
# Typographical Mistakes (Name column)
df['Name'].head(10)
```

Output:

```
0       James Taylor
2       Sarah Connor
4      Michael Scott
6        Emma Watson
8           John Doe
10      Lisa Simpson
12      Robert Brown
14          Anna Lee
16      David Miller
18     Sophie Turner
Name: Name, dtype: str
```

```
# Standardize names: Title case, we can use .str.lower() for case-insensitive matching.
df['Name'] = df['Name'].str.title().str.strip()

# Display the first few rows of the 'Name' column after cleaning to confirm that all values are standardized
df['Name'].head(10)
```

Output:

```
0       James Taylor
2       Sarah Connor
4      Michael Scott
6        Emma Watson
8           John Doe
10      Lisa Simpson
12      Robert Brown
14          Anna Lee
16      David Miller
18     Sophie Turner
Name: Name, dtype: str
```

1.1.11. Contradictory Data

- Son age < father age (truth)
- Son age > father age (not truth remove this contradictory inconsistency)

```
# Contradictory Data (Already handled in 1.1.8)
```

■ Final Clean Data Set Summary

```
df.shape
```

Output:

```
(15, 11)
```

```
df.dtypes
```

Output:

```
Name                   str
Age                  int64
Gender                 str
Country                str
DateOfBirth            str
IsActive               str
Price              float64
Salary             float64
RegistrationDate       str
FatherAge            int64
SonAge               int64
dtype: object
```

```
df.isna().sum()
```

Output:

```
Name                0
Age                 0
Gender              0
Country             0
DateOfBirth         0
IsActive            0
Price               0
Salary              0
RegistrationDate    0
FatherAge           0
SonAge              0
dtype: int64
```

```
df.head(5)
```

Output:

	Name	Age	Gender	Country	DateOfBirth	IsActive	Price	Salary	RegistrationDate	FatherAge	SonAge
0	James Taylor	27	male	United States	1999-01-15	true	100.0	50000.0	1/10/2020	55	25
2	Sarah Connor	30	female	United States	1996-05-20	true	100.0	60000.0	6/15/2019	58	28
4	Michael Scott	16	male	United States	2010-10-03	false	85.5	60000.0	3/1/2021	30	25
6	Emma Watson	28	female	United Kingdom	1998-08-25	true	200.0	60000.0	12/20/2018	62	20
8	John Doe	47	male	Canada	1979-11-30	true	150.0	55000.0	7/22/2017	70	45

★ Upload Cleaned Dataset Directly to GitHub

```python
import requests # for making HTTP requests
import base64 # for encoding/decoding base64 data
import json # for handling JSON data

# GitHub repository details
# ⚠ REPLACE THIS WITH YOUR GITHUB USERNAME
GITHUB_USERNAME = "demo_user"
# ⚠ REPLACE THIS WITH YOUR REPOSITORY NAME
REPO_NAME = "machine_learning_1.0"
# ⚠ REPLACE THIS WITH YOUR FILE PATH
FILE_PATH = "files_and_datasets/f_ds5_II/consistent_data.csv"
# ⚠ REPLACE THIS WITH YOUR BRANCH NAME
BRANCH = "main"

# Convert DataFrame to CSV string
csv_content = df.to_csv(index=False)

# Encode content to base64 (required by GitHub API)
content_encoded = base64.b64encode(csv_content.encode()).decode()

# GitHub API URL
api_url =
f"https://api.github.com/repos/{GITHUB_USERNAME}/{REPO_NAME}/cont
ents/{FILE_PATH}"
```

- **Generate Token**
  - Open this link: https://github.com/settings/tokens/new
  - Confirm Access via "Use GitHub Mobile or "Send a code via email"
  - Note "demo_note"
  - Expiration "No Expiration", but it's upto you.
  - Tick on "repo"
  - Click on "Generate token"
  - "Copy Token"

```python
# You need to provide your GitHub Personal Access Token here
# ⚠ REPLACE THIS WITH YOUR TOKEN
GITHUB_TOKEN = "demo_token"

# Request headers
headers = {
    "Authorization": f"token {GITHUB_TOKEN}",
    "Accept": "application/vnd.github.v3+json"
}

# Request body
data = {
    "message": "Add cleaned dataset - consistent_data.csv",
    "content": content_encoded,
    "branch": BRANCH
}

# Check if file already exists (to get SHA for update)
try:
    check_response = requests.get(api_url, headers=headers)
    if check_response.status_code == 200:
        # File exists, need SHA to update
        sha = check_response.json()["sha"]
        data["sha"] = sha
except:
    print("Creating new file...")

# Upload/Update file
response = requests.put(api_url, headers=headers,
data=json.dumps(data))

if response.status_code in [200, 201]:
    # File location:
    print(f"   {api_url.replace('api.github.com/repos',
        'github.com').replace('/contents/', '/blob/main/')}")
    # Raw file URL:
    print(f"   https://raw.githubusercontent.com/{GITHUB_USERNAME
         }/{REPO_NAME}/main/{FILE_PATH}")
    # You can now load it using:
    print(f'   df =
        pd.read_csv("https://raw.githubusercontent.com/{GITHUB_US
        ERNAME}/{REPO_NAME}/main/{FILE_PATH}")')
else:
    # Upload failed!"
    print(f"Status code: {response.status_code}")
    print(f"Response: {response.json()}")
```

Output:

```
https://github.com/tabassumgulfaraz-ds/machine_learning_1.0/blob/main/files_and_datasets/f_ds5_II/consistent_data.csv

https://raw.githubusercontent.com/tabassumgulfaraz-ds/machine_learning_1.0/main/files_and_datasets/f_ds5_II/consistent_data.csv

df = pd.read_csv(https://raw.githubusercontent.com/tabassumgulfaraz-ds/machine_learning_1.0/main/files_and_datasets/f_ds5_II/consistent_data.csv)
```