Problem Set 1: Analysis of racial disparities in felony sentencing, Part 1

0. Load packages and imports

In [42]:
```
import pandas as pd
import numpy as np
import re

## can add others if you need them

## repeated printouts
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

0.1: Load the data (0 points)

Load the data from `sentencing_asof0405.csv`

- *Notes*: You may receive a warning about mixed data types upon import; feel free to ignore

In [43]:
```
df = pd.read_csv("pset1_inputdata/sentencing_asof0405.csv")
```

```
C:\Users\sharp\AppData\Local\Temp\ipykernel_6340\2150987340.py:1: DtypeWarning: Colu
mns (10,11,14,25) have mixed types. Specify dtype option on import or set low_memory
=False.
  df = pd.read_csv("pset1_inputdata/sentencing_asof0405.csv")
```

0.2: Print head, dimensions, info (0 points)

In [44]:
```
df.head()
df.shape
df.info()
```

Out[44]:

	CASE_ID	CASE_PARTICIPANT_ID	RECEIVED_DATE	OFFENSE_CATEGORY	PRIMARY_C
0	149765331439	175691153649	8/15/1984 12:00:00 AM	PROMIS Conversion	
1	149765331439	175691153649	8/15/1984 12:00:00 AM	PROMIS Conversion	
2	149765331439	175691153649	8/15/1984 12:00:00 AM	PROMIS Conversion	
3	149765331439	175691153649	8/15/1984 12:00:00 AM	PROMIS Conversion	
4	149765331439	175691153649	8/15/1984 12:00:00 AM	PROMIS Conversion	

5 rows × 41 columns

Out[44]:

```
(248146, 41)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 248146 entries, 0 to 248145
Data columns (total 41 columns):
 #   Column                             Non-Null Count   Dtype  
---  ------                             --------------   -----  
 0   CASE_ID                            248146 non-null  int64  
 1   CASE_PARTICIPANT_ID                248146 non-null  int64  
 2   RECEIVED_DATE                      248146 non-null  object 
 3   OFFENSE_CATEGORY                   248146 non-null  object 
 4   PRIMARY_CHARGE_FLAG                248146 non-null  bool   
 5   CHARGE_ID                          248146 non-null  int64  
 6   CHARGE_VERSION_ID                  248146 non-null  int64  
 7   DISPOSITION_CHARGED_OFFENSE_TITLE  248146 non-null  object 
 8   CHARGE_COUNT                       248146 non-null  int64  
 9   DISPOSITION_DATE                   248146 non-null  object 
 10  DISPOSITION_CHARGED_CHAPTER        248146 non-null  object 
 11  DISPOSITION_CHARGED_ACT            242771 non-null  object 
 12  DISPOSITION_CHARGED_SECTION        242771 non-null  object 
 13  DISPOSITION_CHARGED_CLASS          248127 non-null  object 
 14  DISPOSITION_CHARGED_AOIC           248122 non-null  object 
 15  CHARGE_DISPOSITION                 248146 non-null  object 
 16  CHARGE_DISPOSITION_REASON          904 non-null     object 
 17  SENTENCE_JUDGE                     247404 non-null  object 
 18  SENTENCE_COURT_NAME                246761 non-null  object 
 19  SENTENCE_COURT_FACILITY            246216 non-null  object 
 20  SENTENCE_PHASE                     248146 non-null  object 
 21  SENTENCE_DATE                      248146 non-null  object 
 22  SENTENCE_TYPE                      248146 non-null  object 
 23  CURRENT_SENTENCE_FLAG              248146 non-null  bool   
 24  COMMITMENT_TYPE                    246464 non-null  object 
 25  COMMITMENT_TERM                    246434 non-null  object 
 26  COMMITMENT_UNIT                    246434 non-null  object 
 27  LENGTH_OF_CASE_in_Days             229126 non-null  float64
 28  AGE_AT_INCIDENT                    238359 non-null  float64
 29  RACE                               246879 non-null  object 
 30  GENDER                             247337 non-null  object 
 31  INCIDENT_CITY                      228745 non-null  object 
 32  INCIDENT_BEGIN_DATE                239122 non-null  object 
 33  INCIDENT_END_DATE                  22008 non-null   object 
 34  LAW_ENFORCEMENT_AGENCY             239405 non-null  object 
 35  LAW_ENFORCEMENT_UNIT               76408 non-null   object 
 36  ARREST_DATE                        242981 non-null  object 
 37  FELONY_REVIEW_DATE                 171907 non-null  object 
 38  FELONY_REVIEW_RESULT               171907 non-null  object 
 39  ARRAIGNMENT_DATE                   229126 non-null  object 
 40  UPDATED_OFFENSE_CATEGORY           248146 non-null  object 
dtypes: bool(2), float64(2), int64(5), object(32)
memory usage: 74.3+ MB
```

Part 1: data cleaning/interpretation

1.1: Understanding the unit of analysis (5 points)

- Print the number of unique values for the following columns. Do so in a way that avoids copying/pasting code for the three:
    - Cases (`CASE_ID`)
    - People in that case (`CASE_PARTICIPANT_ID`)
    - Charges (`CHARGE_ID`)
- Write a couple sentences on the following and show an example of each (e.g., a case involving multiple people):
    - Why there are more unique people than unique cases?
    - Why there are more unique charges than unique people?
- Print the mean and median number of charges per case
- Print the mean and median number of participants per case
- Does the data seem to enable us to follow the same defendant across different cases they're charged in? Write 1 sentence in support of your conclusion.

In [45]:
```
# No. of unique values in CASE_ID, CASE_PARTICIPANT_ID, CHARGE_ID, in a way that av
unique_counts = df[["CASE_ID","CASE_PARTICIPANT_ID","CHARGE_ID"]].nunique()
print(unique_counts)
```

```
CASE_ID                197519
CASE_PARTICIPANT_ID    211977
CHARGE_ID              229015
dtype: int64
```

There are more unique people than unique cases because each unique cases can be committed by one or more unique person. For example, a robbery case involving 2 people.

In [46]:
```
# Find a case with multiple participants
participants_counts = df.groupby("CASE_ID")["CASE_PARTICIPANT_ID"].nunique()
multi_participants_ex = participants_counts[participants_counts > 1].index[0]

# Show the participants for that case
participants = df[df["CASE_ID"] == multi_participants_ex]["CASE_PARTICIPANT_ID"].un
print(f"EXAMPLE: CASE_ID {multi_participants_ex} has participants: {participants}")
```

```
EXAMPLE: CASE_ID 166402790922 has participants: [144234439761 144234534133]
```

There are more unique charges than unique people becuase one person can be charged with multiple counts of crime. For example, if someone tried to murder their pregnant wife and robbed them of their possesions then that would be intentional homicide of unborn child, murder, and armed robbery.

In [47]:
```
# Find participants with more than one unique charge
participant_charge_counts = df.groupby("CASE_PARTICIPANT_ID")["CHARGE_ID"].nunique(
multi_charge_participant = participant_charge_counts[participant_charge_counts > 1]

# Show the charges for that participant
```

```
charges = df[df["CASE_PARTICIPANT_ID"] == multi_charge_participant]["DISPOSITION_CH
print(f"EXAMPLE: CASE_PARTICIPANT_ID {multi_charge_participant} has charges: {charg
```

```
EXAMPLE: CASE_PARTICIPANT_ID 97581722610 has charges: ['INT HOMI OF UNBORN CHILD=38-
9-1.2' 'MURDER=720-5\\9-1(A)(1-3)'
 'ARMED ROBBERY=720-5\\18-2(A)']
```

In [48]:

```
# Mean and median number of charges per case
charges_per_case = df.groupby("CASE_ID")["CHARGE_ID"].nunique()
print("Mean number of charges per case:", charges_per_case.mean())
print("Median number of charges per case:", charges_per_case.median())

# Mean and median number of participants per case
participants_per_case = df.groupby("CASE_ID")["CASE_PARTICIPANT_ID"].nunique()
print("Mean number of participants per case:", participants_per_case.mean())
print("Median number of participants per case:", participants_per_case.median())
```

```
Mean number of charges per case: 1.1594580774507768
Median number of charges per case: 1.0
Mean number of participants per case: 1.0731980214561636
Median number of participants per case: 1.0
```

The data does not seem to enable us to follow the same defendant across different cases because `CASE_PARTICIPANT_ID` is unique to each case-participant combination, not to each defendant. There is no persistent defendant ID across different cases in the visible columns.

1.2.1: Which offense is final? (3 points)

- First, read the data documentation link and summarize in your own words the differences between `OFFENSE_CATEGORY` and `UPDATED_OFFENSE_CATEGORY`
- Construct an indicator `is_changed_offense` that's True for case-participant-charge observations (rows) where there's a difference between the original charge (offense category) and the most current charge (updated offense category). What are some of the more common changed offenses? (can just print result of sort_values based on original offense category)
- Print one example of a changed offense from one of these categories and comment on what the reason may be

OFFENSE_CATEGORY is the case's initial broad offense label before charges are finalized, while UPDATED_OFFENSE_CATEGORY is the later, revised label recalculated to match the case's primary/most severe charge.

In [49]:

```
# Create indicator for changed offense (True if original differs from updated)
is_changed_offense = df["OFFENSE_CATEGORY"].fillna("") != df["UPDATED_OFFENSE_CATEG

# Add to datafram
df["is_changed_offense"] = is_changed_offense
```

```
# Count the charged offenses based on original offense category
charged_offenses_sorted = df[df["is_changed_offense"]].groupby("OFFENSE_CATEGORY").
print(f"Total rows with changed offense: {is_changed_offense.sum()}")
print("\nMost common changed offenses (by original offense category):")
print(charged_offenses_sorted)
```

```
Total rows with changed offense: 35865

Most common changed offenses (by original offense category):
                  OFFENSE_CATEGORY  count
61               PROMIS Conversion   6394
33                             DUI   3896
81    UUW - Unlawful Use of Weapon   2155
60                   Other Offense   2125
2               Aggravated Battery   1927
..                             ...    ...
63                         Perjury      4
70                    Prostitution      3
22         Benefit Recipient Fraud      2
29       Compelling Gang Membership     2
85                 Violate Bail Bond      2

[88 rows x 2 columns]
```

In [50]:
```
# Print one example of a changed offense from a common category
common_offcat_example = df[df["is_changed_offense"] & (df["OFFENSE_CATEGORY"] != "P
print("Example of a changed offense:")
print(common_offcat_example.to_string(index=False))
```

```
Example of a changed offense:
OFFENSE_CATEGORY UPDATED_OFFENSE_CATEGORY  CASE_PARTICIPANT_ID   CHARGE_ID
Attempt Homicide          Domestic Battery         203478864452 89337340966
```

The change in attempt homicide to domestic battery is based on the intent of the criminal and/or insufficient evidence, where battery is the intent to injury and homicide is intent to kill.

1.2.2: Simplifying the charges (5 points)

Using the field (`UPDATED_OFFENSE_CATEGORY`), create a new field, `simplified_offense_derived` , that simplifies the many offense categories into broader buckets using the following process:

First, combine all offenses beginning with "Aggravated" into a single category without that prefix (e.g., Aggravated Battery and Battery just becomes Battery)

Then:

- Combine all offenses with arson into a single arson category (`Arson`)
- Combine all offenses with homicide into a single homicide category (`Homicide`)

- Combine all offenses with vehicle/vehicular in the name into a single vehicle category (`Vehicle-related`)
- Combine all offenses with battery in the name into a single battery category (`Battery`)

Try to do so efficiently (e.g., write a function and apply to a column, rather than edit the variable repeatedly in separate line for each recoded offense)

Print the difference between the # of unique offenses in the original `UPDATED_OFFENSE_CATEGORY` field and the # of unique offenses in your new `simplified_offense_derived` field

In [51]:
```
# Build a function that first identifies aggravated charges, removes the prefix, an
def simplify_offense_aggravated(offense):
    if pd.isna(offense):
        return np.nan
    offense_lower = str(offense).lower().strip()
    # Remove leading "Aggravated" (case-insensitive)
    if offense_lower.startswith("aggravated "):
        offense_lower = offense_lower[len("aggravated "):].strip()
    if "arson" in offense_lower:
        return "Arson"
    elif "homicide" in offense_lower:
        return "Homicide"
    elif "vehic" in offense_lower:
        return "Vehicle-related"
    elif "battery" in offense_lower:
        return "Battery"
    else:
        return offense_lower

# Applying to a column
df["simplified_offense_derived"] = df["UPDATED_OFFENSE_CATEGORY"].apply(simplify_of

# Show the difference between the number of unique offenses before and after simpli
original_unique = df["UPDATED_OFFENSE_CATEGORY"].nunique(dropna=False)
simplified_unique = df["simplified_offense_derived"].nunique(dropna=False)
print(f"Unique offenses before simplification: {original_unique}")
print(f"Unique offenses after simplification: {simplified_unique}")
print(f"Difference: {original_unique - simplified_unique}")
```

```
Unique offenses before simplification: 79
Unique offenses after simplification: 65
Difference: 14
```

1.3: Cleaning additional variables (10 points)

Clean the following variables; make sure to retain the original variable in data and use the derived suffix so it's easier to pull these cleaned out variables later (e.g., `age_derived`) to indicate this was a transformation

- Race: create True/false indicators for `is_black_derived` (Black only or mixed race with hispanic), Non-Black Hispanic, so either hispanic alone or white hispanic (`is_hisp_derived`), White non-hispanic (`is_white_derived`), or none of the above (`is_othereth_derived`)
- Gender: create a boolean true/false indicator for `is_male_derived` (false is female, unknown, or other)
- Age at incident: you notice outliers like 130-year olds. Winsorsize the top 0.01% of values to be equal to the 99.99th percentile value pre-winsorization. Call this `age_derived`
- Create `sentenceymd_derived` that's a version of `SENTENCE_DATE` converted to datetime format. Also create a rounded version, `sentenceym_derived` , that's rounded down to the first day of the month (e.g., `1/5/2016` would become `1/1/2016` and `3/27/2018` would become `3/1/2018`)
  - Hint: all timestamps are midnight so u can strip in conversion. For full credit, before converting, you notice that some of the years have been mistranscribed (e.g., 291X or 221X instead of 201X). Programatically fix those (eg 2914 -> 2014). Even after cleaning, there will still be some that are after the year 2021 that we'll filter out later. For partial credit, you can ignore the timestamps that cause errors and set errors = "coerce" within `pd.to_datetime()` to allow the conversion to proceed.
- Sentencing judge: create an identifier (`judgeid_derived`) for each unique judge (`SENTENCE_JUDGE`) structured as judge_1, judge_2...., with the order determined by sorting the judges (will sort on fname then last). When finding unique judges, there are various duplicates we could weed out --- for now, just focus on (1) the different iterations of Doug/Douglas Simpson, (2) the different iterations of Shelley Sutker (who appears both with her maiden name and her hyphenated married name).
  - Hint: due to mixed types, you may need to cast the `SENTENCE_JUDGE` var to a diff type to sort

After finishing, print a random sample of 10 rows (data.sample(n = 10)) with the original and cleaned columns for the relevant variables to validate your work

In [52]:
```
# Print all unique values of RACE column for inspection
print("All unique values in RACE column:")
print(sorted(df["RACE"].dropna().unique()))
```

```
All unique values in RACE column:
['ASIAN', 'American Indian', 'Asian', 'Biracial', 'Black', 'HISPANIC', 'Unknown', 'W
hite', 'White [Hispanic or Latino]', 'White/Black [Hispanic or Latino]']
```

In [53]:
```
# Race Indicators
race_str = df["RACE"].str.upper().fillna("")

df["is_black_derived"] = (
```

```python
    race_str.isin(["BLACK"]) |
    (race_str.str.contains("BLACK") & race_str.str.contains("HISPANIC"))
)

df["is_hisp_derived"] = (
    race_str.str.contains("HISPANIC") | race_str.str.contains("LATINO")
) & ~df["is_black_derived"]

df["is_white_derived"] = (
    race_str.str.contains("WHITE") &
    ~race_str.str.contains("HISPANIC") &
    ~race_str.str.contains("LATINO") &
    ~df["is_black_derived"]
)

df["is_othereth_derived"] = ~(
    df["is_black_derived"] | df["is_hisp_derived"] | df["is_white_derived"]
)
```

In [54]:
```python
# Output code for race indicators

race_cols = ["RACE", "is_black_derived", "is_hisp_derived", "is_white_derived", "is

def show_race_examples(df):
    # Find one example for each derived group
    examples = []
    ex_black = df[df["is_black_derived"]].head(1)
    if not ex_black.empty:
        examples.append(ex_black)
    ex_hisp = df[df["is_hisp_derived"]].head(1)
    if not ex_hisp.empty:
        examples.append(ex_hisp)
    ex_white = df[df["is_white_derived"]].head(1)
    if not ex_white.empty:
        examples.append(ex_white)
    ex_other = df[df["is_othereth_derived"]].head(1)
    if not ex_other.empty:
        examples.append(ex_other)
    if examples:
        display(pd.concat(examples)[race_cols])
    else:
        print("No examples found for the derived race indicators.")

print("Targeted examples for each race indicator:")
show_race_examples(df)

print("\nRace indicator value counts:")
for col in race_cols[1:]:
    print(df[col].value_counts())
    print()
```

```
Targeted examples for each race indicator:
```

	RACE	is_black_derived	is_hisp_derived	is_white_derived	is_othereth_derived
0	Black	True	False	False	False
18	White [Hispanic or Latino]	False	True	False	False
26	White	False	False	True	False
41	NaN	False	False	False	True

```
Race indicator value counts:
is_black_derived
True     165661
False     82485
Name: count, dtype: int64

is_hisp_derived
False    204325
True      43821
Name: count, dtype: int64

is_white_derived
False    212785
True      35361
Name: count, dtype: int64

is_othereth_derived
False    244843
True       3303
Name: count, dtype: int64
```

In [ ]:

```
print(df["GENDER"].str.upper().unique())

# Gender Indicator: only true if MALE
gender_upper = df["GENDER"].str.upper().fillna("")
df["is_male_derived"] = gender_upper.isin(["MALE"])
print(df["is_male_derived"].value_counts())
```

```
['MALE' 'FEMALE' nan 'MALE NAME, NO GENDER GIVEN' 'UNKNOWN GENDER'
 'UNKNOWN']
is_male_derived
True     217610
False     30536
Name: count, dtype: int64
```

In [56]:

```
# Age at Incident Winsorization (Basically capping the 99.99%)
age_col = "AGE_AT_INCIDENT"
percentile_9999 = df[age_col].quantile(0.9999)
df["age_derived"] = df[age_col].clip(upper=percentile_9999)
print(df["age_derived"].describe())
```

```
count    238359.000000
mean         32.302611
std          11.779161
min          17.000000
25%          23.000000
50%          29.000000
75%          40.000000
max          81.000000
Name: age_derived, dtype: float64
```

In [57]:

```python
# Sentence Date Reformating to datetime

# Function to fix mistranscribed years
def fix_year(date_str):
    if pd.isna(date_str):
        return date_str

    s = str(date_str).strip()
    if not s:
        return s

    parts = s.split()
    date_part = parts[0]
    rest = " ".join(parts[1:])

    # Only handle slash dates like MM/DD/YYYY
    if "/" not in date_part:
        return s

    d = date_part.split("/")
    if len(d) != 3:
        return s

    mm, dd, yyyy = d[0], d[1], d[2]

    # Fix 4-digit years where digits are mistranscribed
    if yyyy.isdigit() and len(yyyy) == 4:
        year_int = int(yyyy)
        if year_int > 2025 and year_int < 9999:
            yyyy = "20" + yyyy[2:]

    fixed_date = f"{mm}/{dd}/{yyyy}"
    return f"{fixed_date} {rest}".strip()


test_dates = [
    "05/01/2914 12:00:00 AM",
    "12/31/2218 12:00:00 AM",
    "01/01/2119 12:00:00 AM",
    "06/15/2020 12:00:00 AM",
    "07/20/2015 12:00:00 AM",
    "03/10/2512 12:00:00 AM",
    "03/10/3212 12:00:00 AM",
    None,
    "random text"
]
```

```python
# Print before and after for each test case
for d in test_dates:
    print(f"Original: {d}  -->  Fixed: {fix_year(d)}")

# Apply the fix function and convert to datetime
df["SENTENCE_DATE_cleaned"] = df["SENTENCE_DATE"].apply(fix_year)
df["sentenceymd_derived"] = pd.to_datetime(df["SENTENCE_DATE_cleaned"], format="%m/

# Round down to first day of month
df["sentenceym_derived"] = df["sentenceymd_derived"].dt.to_period("M").dt.start_tim

# Drop column no longer needed since transferred to sentenceymd_derived
df.drop(columns=["SENTENCE_DATE_cleaned"], inplace=True)
```

```
Original: 05/01/2914 12:00:00 AM  -->  Fixed: 05/01/2014 12:00:00 AM
Original: 12/31/2218 12:00:00 AM  -->  Fixed: 12/31/2018 12:00:00 AM
Original: 01/01/2119 12:00:00 AM  -->  Fixed: 01/01/2019 12:00:00 AM
Original: 06/15/2020 12:00:00 AM  -->  Fixed: 06/15/2020 12:00:00 AM
Original: 07/20/2015 12:00:00 AM  -->  Fixed: 07/20/2015 12:00:00 AM
Original: 03/10/2512 12:00:00 AM  -->  Fixed: 03/10/2012 12:00:00 AM
Original: 03/10/3212 12:00:00 AM  -->  Fixed: 03/10/2012 12:00:00 AM
Original: None  -->  Fixed: None
Original: random text  -->  Fixed: random text
```

In [58]:

```python
# Judge ID Creation

# Function to standardize judge names before creating mapping
def standardize_judge(j):
    if pd.isna(j):
        return j

    j = str(j).strip().lower()

    if j =="":
        return pd.NA

    if ("simpson" in j) and ("doug" in j or "douglas" in j):
        return "Douglas Simpson"
    if ("sutker" in j) and ("shelley" in j):
        return "Shelley Sutker"

    return j.title()  # Capitalize names for consistency

# Apply standardization
df["judge_cleaned"] = df["SENTENCE_JUDGE"].astype(str).apply(standardize_judge)

# Get unique judges, remove missing values and null values, and sort them
unique_judges = df["judge_cleaned"].dropna().unique()
unique_judges_sorted = sorted(unique_judges)

# Create a mapping from judge name to ID
judge_mapping = {judge: f"judge_{i+1}" for i, judge in enumerate(unique_judges_sort
df["judgeid_derived"] = df["judge_cleaned"].map(judge_mapping)

# Dropped column due to mapping on to judgeid_derived
```

```python
df.drop(columns=["judge_cleaned"], inplace=True)

# Show mapping for Doug/Douglas Simpson and Shelley Sutker variations
simpson_mask = df["SENTENCE_JUDGE"].astype(str).str.contains("Simpson")
sutker_mask = df["SENTENCE_JUDGE"].astype(str).str.contains("Sutker")

print("Doug/Douglas Simpson mapping:")
print(df.loc[simpson_mask, ["SENTENCE_JUDGE", "judgeid_derived"]].drop_duplicates()

print("\nShelley Sutker variations mapping:")
print(df.loc[sutker_mask, ["SENTENCE_JUDGE", "judgeid_derived"]].drop_duplicates())

print("\nGeneral sample of judge mappings:")
print(df[["SENTENCE_JUDGE", "judgeid_derived"]].drop_duplicates().head(10))
```

```
Doug/Douglas Simpson mapping:
         SENTENCE_JUDGE judgeid_derived
1115      Doug  Simpson        judge_71
1621  Douglas J Simpson        judge_71

Shelley Sutker variations mapping:
             SENTENCE_JUDGE judgeid_derived
131          Shelley  Sutker       judge_281
142  Shelley  Sutker-Dermer       judge_281

General sample of judge mappings:
          SENTENCE_JUDGE judgeid_derived
0          John  Mannion       judge_140
4       Clayton Jay Crane        judge_41
11         James L Rhodes       judge_114
12        Thomas V Gainer       judge_312
18           Kay M Hanlon       judge_163
19       William J Kunkle       judge_333
20           Evelyn B Clay        judge_84
26   Timothy Joseph Joyce       judge_316
27        Steven J Goebel       judge_289
29          Carol M Howard        judge_32
```

In [61]:
```python
# Group the columns that are needed for validation
validation_col = ["RACE", "is_black_derived", "is_hisp_derived", "is_white_derived"
                  "GENDER", "is_male_derived",
                  "AGE_AT_INCIDENT", "age_derived",
                  "SENTENCE_DATE", "sentenceymd_derived", "sentenceym_derived",
                  "SENTENCE_JUDGE", "judgeid_derived"]

print("Random sample of 10 rows with original vs cleaned columns")
df.sample(n=10)[validation_col]
```

```
Random sample of 10 rows with original vs cleaned columns
```

Out[61]:

	RACE	is_black_derived	is_hisp_derived	is_white_derived	is_othereth_derived	G
26970	White	False	False	True	False	
212818	Black	True	False	False	False	
228985	Black	True	False	False	False	
59326	White [Hispanic or Latino]	False	True	False	False	
86994	Black	True	False	False	False	
1831	Black	True	False	False	False	
197013	Black	True	False	False	False	
75650	White	False	False	True	False	
79633	Black	True	False	False	False	
184935	White [Hispanic or Latino]	False	True	False	False	

1.4: Subsetting rows to analytic dataset (5 points)

You decide based on the above to simplify things in the following ways:

- Subset to cases where only one participant is charged, since cases with >1 participant might have complications like plea bargains/informing from other participants affecting the sentencing of the focal participant
- To go from a participant-case level dataset, where each participant is repeated across charges tied to the case, to a participant-level dataset, where each participant has one charge, subset to a participant's primary charge and their current sentence (`PRIMARY_CHARGE_FLAG` is True and `CURRENT_SENTENCE_FLAG` is True). Double check that this worked by confirming there are no longer multiple charges for the same case-participant

- Filter out observations where judge is nan or nonsensical (indicated by is.null or equal to FLOOD)

- Subset to sentencing date between 01-01-2012 and 04-05-2021 (inclusive)

After completing these steps, print the number of rows in the data

In [60]:
```
print("Original DataFrame shape:", df.shape, "\n")
# Subset cases where only 1 participant is charged
single_participant_cases = participants_counts[participants_counts == 1].index
df_analytic = df[df["CASE_ID"].isin(single_participant_cases)].copy()
print("After subsetting to single-participant cases.", df_analytic.shape, "\n")

# Reduce to a participant-level dataset with only primary charges and current sente
focal = df_analytic["PRIMARY_CHARGE_FLAG"] & df_analytic["CURRENT_SENTENCE_FLAG"]
df_analytic = df_analytic.loc[focal].copy()
print("After subsetting to primary charges with current sentences:", df_analytic.sh

# Confirm no multiple charges for the same case-participant
charges_per_case_participant = df_analytic.groupby(["CASE_ID", "CASE_PARTICIPANT_ID
print("Max unique charges per case-participant after subsetting:", charges_per_case
print("Min unique charges per case-participant after subsetting:", charges_per_case

# Filter out observations where judge is nan or nonsensical
j = df_analytic["SENTENCE_JUDGE"]
judge_ok = j.notna() & j.astype(str).str.strip().ne("") & j.astype(str).str.strip()
df_analytic = df_analytic.loc[judge_ok].copy()

print("Drop n/a judges:", df_analytic.shape, "\n")

# Subset to sentencing date between 01-01-2012 and 04-05-2021 inclusive
start_date = pd.Timestamp("2012-01-01")
end_date = pd.Timestamp("2021-04-05")

date_ok = df_analytic["sentenceymd_derived"].between(start_date, end_date, inclusiv
df_analytic = df_analytic.loc[date_ok].copy()

# Print number of rows after subsetting
print("Number of rows in analytic dataset after subsetting:", len(df_analytic))

# Sanity Check
print("\nSanity Check of Analytic Dataset:")

print("Unique CASE_PARTICIPANT_ID in analytic dataset:", df_analytic["CASE_PARTICIP
print("Unique CASE_ID in analytic dataset:", df_analytic["CASE_ID"].nunique())
```

```
Original DataFrame shape: (248146, 52)

After subsetting to single-participant cases. (216252, 52)

After subsetting to primary charges with current sentences: (152900, 52)

Max unique charges per case-participant after subsetting: 1
Min unique charges per case-participant after subsetting: 1

Drop n/a judges: (152449, 52)

Number of rows in analytic dataset after subsetting: 135165

Sanity Check of Analytic Dataset:
Unique CASE_PARTICIPANT_ID in analytic dataset: 135165
Unique CASE_ID in analytic dataset: 135165
```