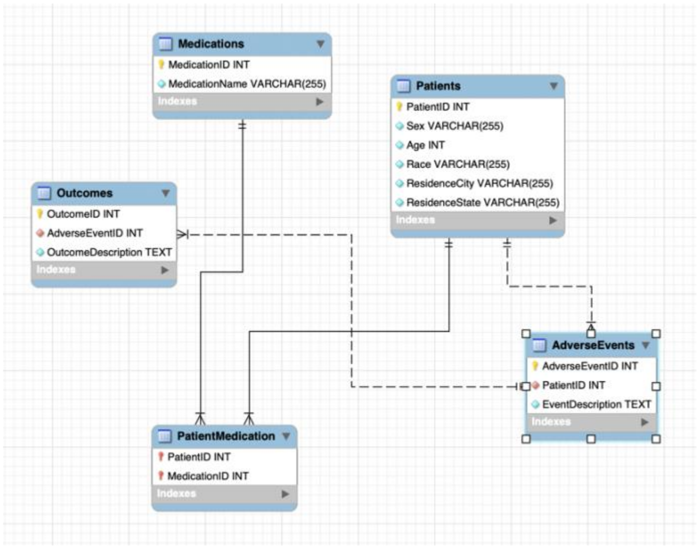
DATABASE DESIGN:

REQUIRED QUESTION:

Detail your ERD design, including an explanation of each table.



Patients Table:

- PatientID: Primary Key (PK), uniquely identifies each patient.
- Sex: Gender of the patient (e.g., Male, Female).
- Race: Ethnicity or race of the patient.
- ResidenceCity: City where the patient resides.
- ResidenceState: State where the patient resides.

The Patients table stores information about individual patients, including their demographic details such as gender, race, and residence location.
Cardinality: One patient can have many adverse events, medications, and outcomes

Medications Table:

- MedicationID: PK, uniquely identifies each medication.
- MedicationName: Name of the medication.

The Medications table contains a list of medications prescribed or administered to patients, identified by a unique MedicationID. It includes the names of the medications.
Cardinality: One medication can be associated with many adverse events and patients.

AdverseEvents Table:

- AdverseEventID: PK, uniquely identifies each adverse event.
- PatientID: Foreign Key (FK) referencing Patients.PatientID, indicates the patient affected by the adverse event.
- EventDescription: Description of the adverse event experienced by the patient.

The AdverseEvents table records instances of adverse events experienced by patients, linking them to specific medications. Each adverse event is identified by a unique AdverseEventID.
Cardinality: One adverse event is specific to one patient and one medication.

Outcomes Table:

- OutcomeID: PK, uniquely identifies each outcome.
- PatientID: FK referencing Patients.PatientID, identifies the patient associated with the outcome.
- AdverseEventID: FK referencing AdverseEvents.AdverseEventID, specifies the adverse event related to the outcome.
- OutcomeDescription: Description of the outcome of the adverse event.

The Outcomes table stores information about the outcomes resulting from adverse events experienced by patients. Each outcome is associated with a specific adverse event.
Cardinality: One outcome is specific to one adverse event and one patient.

PatientMedications Table:

- PatientID: FK referencing Patients.PatientID, identifies the patient prescribed the medication.
- MedicationID: FK referencing Medications.MedicationID, specifies the medication prescribed to the patient.

The PatientMedications table represents the many-to-many relationship between patients and medications. It links patients to the medications they are prescribed or have taken.
Cardinality: Many patients can be associated with many medications, indicating a many-to-many relationship between patients and medications.

This ERD design organizes information related to patients, medications, adverse events, and outcomes, facilitating the tracking and analysis of drug-related incidents and patient outcomes.

Entity-Relationship Diagram Analysis

Introduction
This section of the report provides a detailed analysis of the Entity-Relationship Diagram (ERD) employed in the project, focusing on the entities' strengths, cardinalities of relationships, and the integrity of the database design.

Strong Entities
The database design includes two primary strong entities:

- Patients: Identified by a unique `ID`, this entity represents individuals within the healthcare system and is characterized by attributes such as `Age`, `Sex`, `Race`, `ResidenceCity`, and `ResidenceState`. As a strong entity, `Patients` has the capability to exist independently within the database structure.
- Medications: With a unique `Medication_ID`, this entity details the medications that may be prescribed to patients. The `MedicationName` attribute provides the necessary description. Similar to `Patients`, `Medications` is a strong entity that does not rely on other entities for its definition or existence.

Weak Entity
The ERD features one weak entity:
PatientMedication: This associative entity does not have an independent primary key but rather uses a composite key consisting of `PatientID` and `Medication_ID`, which are foreign keys from the `Patients` and `Medications` entities, respectively. The purpose of the `PatientMedication` entity is to manage the many-to-many relationship between patients and their prescribed medications.
Cardinality Analysis

The relationships and cardinalities between the entities are as follows:

- Patients to PatientMedication: Exhibits a one-to-many (1:N) cardinality. Each patient can have multiple associated medication records, while each entry in the `PatientMedication` table references a single patient.
- Medications to PatientMedication: Also displays a one-to-many (1:N) cardinality. A given medication can be linked to several entries in the `PatientMedication` table, yet each of those entries corresponds to just one medication record.
- AdverseEvents to Patients: This relationship is one-to-many (1:N), where a single patient may have several adverse event records. Conversely, each adverse event is specific to one patient.
- Outcomes to AdverseEvents: Portrayed as a one-to-one (1:1) relationship, indicating that each adverse event has one corresponding outcome record, although not every adverse event necessarily results in an outcome.

Database Integrity and Normalization
The introduction of the `PatientMedication` entity serves to reinforce database integrity by establishing clear referential links between patients and their medications. It demonstrates a commitment to third normal form (3NF) by eliminating transitive dependencies and repeating groups, hence reducing data redundancy and promoting efficient data management.

ELECTIVE QUESTIONS:

Evaluate your database's normalization. To which normal form is it normalized? If not normalized, propose improvements.
Normalization is a process in database design that organizes the attributes and tables of a relational database to minimize redundancy and dependency. The normalization process aims to reduce data anomalies such as update anomalies, insertion anomalies, and deletion anomalies.
In my database design, let's assess the normalization level:
1. First Normal Form (1NF): Each column contains atomic values, and there are no repeating groups or arrays. All tables in my design meet the requirements of 1NF. Each column contains atomic values, and there are no repeating groups.
2. Second Normal Form (2NF): The table is in 1NF, and all attributes are fully functionally dependent on the primary key. My database design meets the requirements of 2NF. Each non-primary key attribute is dependent on the entire primary key, and there are no partial dependencies.
3. Third Normal Form (3NF): The table is in 2NF, and there are no transitive dependencies. My database design meets the requirements of 3NF as well. There are no transitive dependencies between non-primary key attributes.
Based on the assessment, my database design is normalized up to the third normal form (3NF).

Describe how your database design allows for the addition of new attributes. How would you implement these changes using SQL?
In my database design, adding new attributes is flexible because each table is structured with columns that can accommodate additional attributes as needed. To add a new attribute using SQL, I would use the ALTER TABLE statement. For example, to add a "DateStarted" attribute to the PatientMedications table:

```
ALTER TABLE PatientMedications
ADD DateStarted DATE;
```

Discuss the selection of primary keys in your design, including the use of composite keys (if any):

In my database design, each table has a primary key designated to uniquely identify each record within that table.
Patients Table: The primary key is PatientID, which uniquely identifies each patient in the database.
Medications Table: The primary key is MedicationID, providing a unique identifier for each medication entry.
AdverseEvents Table: AdverseEventID serves as the primary key, ensuring each adverse event record is uniquely identified.
Outcomes Table: OutcomeID is the primary key, uniquely identifying each outcome associated with adverse events.
PatientsMedications Table: In my design, the PatientsMedications table contains two separate primary keys: PatientID and MedicationID. This decision is justified by the fact that PatientsMedications serves as an intermediary table to establish a many-to-many relationship between Patients and Medications. Each entry in this table represents a unique combination of a patient and a medication. Therefore, both PatientID and MedicationID need to be unique within the context of this table to accurately represent the relationships between patients and medications. Having separate primary keys ensures that each patient-medication combination is uniquely identifiable and maintains the integrity of the relationships in the database.

How have you implemented referential integrity? Provide a specific example from your project:
Referential integrity is enforced through foreign key constraints in my database design. For example, in the AdverseEvents table, both PatientID and MedicationID are foreign keys referencing the Patients and Medications tables, respectively. This ensures that a record cannot be inserted into the AdverseEvents table unless the corresponding patient and medication records exist in their respective tables.

DATA ANALYTICS

REQUIRED QUESTION:

Which attributes were used for analytics? Provide SQL queries used to retrieve this data from your database.

For analytics, the following attributes were used:

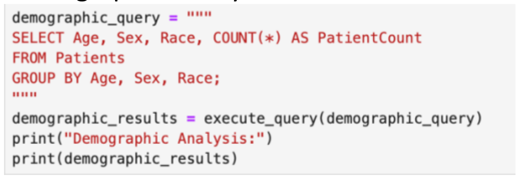
- Patient demographic information (`Age`, `Sex`, `Race`)
- Patient geospatial information (`ResidenceCity`, `ResidenceState`)
- Medication data (`Medication_ID`, `MedicationName`)
- Adverse event data (`AdverseEventID`, `EventDescription`)
- Outcomes data (`OutcomeID`, `OutcomeDescription`)

Analysis done-
1. Demographic Analysis: I explored patient demographic distributions by age, sex, and race using a SQL query that groups the data by these attributes and counts the number of patients within each group. This helped identify which demographics were most affected by drug-related adverse events.
2. Geospatial Analysis: I analyzed the data to uncover geospatial patterns by generating counts of patients within each city and state. This highlighted regions with higher incidences of drug-related issues, pinpointing locations where public health resources might be most needed.
3. Medication Usage Patterns: By running a series of SQL queries, I determined how frequently each medication was associated with adverse events. This was done by counting the number of adverse events linked to each medication ID, revealing the most commonly involved medications in such events.

4. Comorbidity and Polypharmacy Analysis: I assessed the prevalence of multiple medication usage among patients by running queries to count the distinct number of patients taking each medication in combination with Heroin.
5. Detailed Opioid Analysis: I ran a detailed analysis of specific opioids, identifying the most commonly used opioid among the population by executing 17 different SQL queries to count the number of patients for each medication.
6. City-Specific Drug Usage Analysis: I specifically analyzed drug usage data from Connecticut to ascertain that it had the highest drug usage, which pointed towards a larger issue that might require city-level public health interventions.

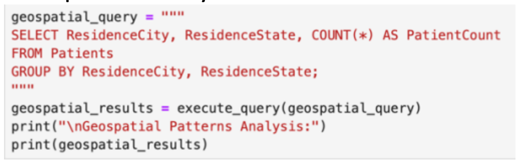
SQL codes used for analysis for the objectives of my project-

Demographic Query:



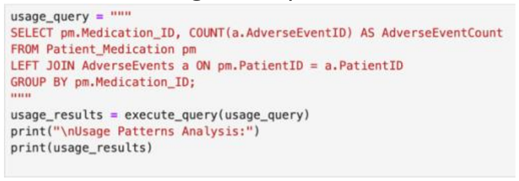
```
demographic_query = """
SELECT Age, Sex, Race, COUNT(*) AS PatientCount
FROM Patients
GROUP BY Age, Sex, Race;
"""
demographic_results = execute_query(demographic_query)
print("Demographic Analysis:")
print(demographic_results)
```

Geospatial Query:



```
geospatial_query = """
SELECT ResidenceCity, ResidenceState, COUNT(*) AS PatientCount
FROM Patients
GROUP BY ResidenceCity, ResidenceState;
"""
geospatial_results = execute_query(geospatial_query)
print("\nGeospatial Patterns Analysis:")
print(geospatial_results)
```

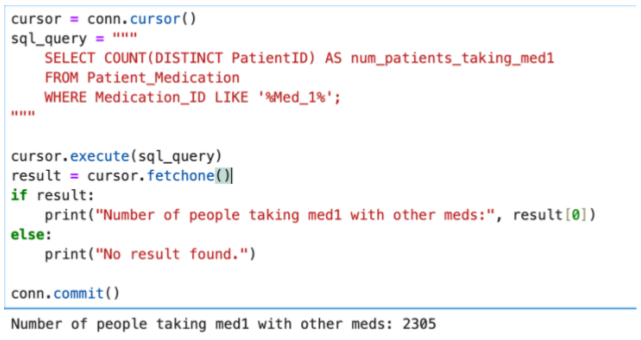
Medication Usage Query:



```
usage_query = """
SELECT pm.Medication_ID, COUNT(a.AdverseEventID) AS AdverseEventCount
FROM Patient_Medication pm
LEFT JOIN AdverseEvents a ON pm.PatientID = a.PatientID
GROUP BY pm.Medication_ID;
"""
usage_results = execute_query(usage_query)
print("\nUsage Patterns Analysis:")
print(usage_results)
```

ELECTIVE QUESTIONS:

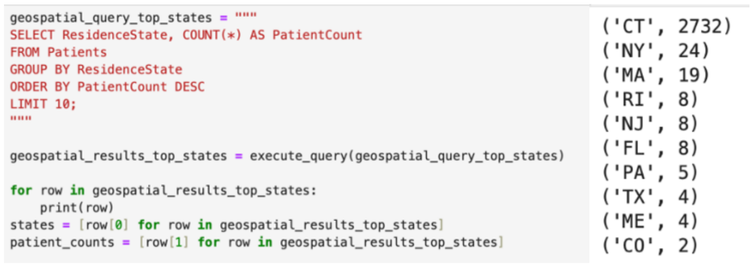
Describe a significant insight or finding from your data analysis. Explain the process of reaching this conclusion.
In the course of my data analysis, I uncovered a crucial insight: the identification of the most prevalently used opioid medication by patients. To achieve this, I executed a series of SQL queries—one for each of the 17 medications in my dataset. These queries counted the distinct number of patients associated with each specific medication, as evidenced by the presence of the medication's ID in the `Patient_Medication` table.



```
cursor = conn.cursor()
sql_query = """
    SELECT COUNT(DISTINCT PatientID) AS num_patients_taking_med1
    FROM Patient_Medication
    WHERE Medication_ID LIKE '%Med_1%';
"""

cursor.execute(sql_query)
result = cursor.fetchone()
if result:
    print("Number of people taking med1 with other meds:", result[0])
else:
    print("No result found.")

conn.commit()
```

```
Number of people taking med1 with other meds: 2305
```

I replicated this query for each medication, from Med_1 through Med_17. By analyzing the counts, I could pinpoint the most widely used medication among the patients.
Furthermore, to assess the prevalence of drug usage across different states, I conducted a geospatial analysis. This involved aggregating the patient data by their residence state.
The query was designed to group the patients by state and then order the results by the patient count in descending order, highlighting the state with the maximum drug usage:



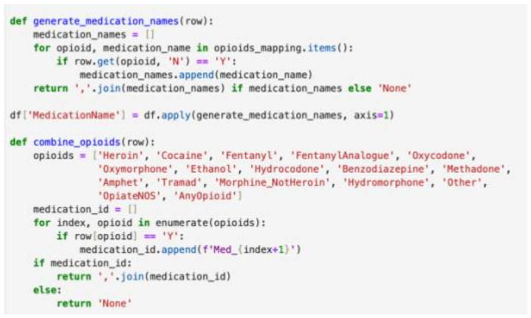
```
geospatial_query_top_states = """
SELECT ResidenceState, COUNT(*) AS PatientCount
FROM Patients
GROUP BY ResidenceState
ORDER BY PatientCount DESC
LIMIT 10;
"""

geospatial_results_top_states = execute_query(geospatial_query_top_states)

for row in geospatial_results_top_states:
    print(row)
states = [row[0] for row in geospatial_results_top_states]
patient_counts = [row[1] for row in geospatial_results_top_states]
```

```
('CT', 2732)
('NY', 24)
('MA', 19)
('RI', 8)
('NJ', 8)
('FL', 8)
('PA', 5)
('TX', 4)
('ME', 4)
('CO', 2)
```

Through this query, I was able to reveal that Connecticut was the state with the highest number of reported drug usage cases, indicating a critical area for public health focus and interventions related to opioid abuse.

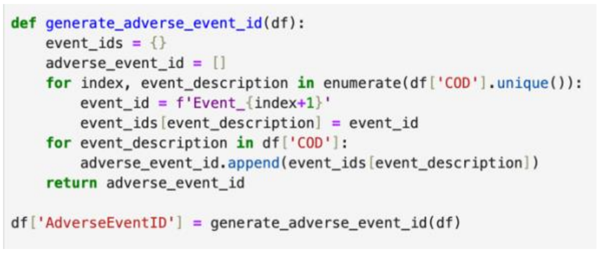
Provide an example of a complex data transformation from your project and its significance.
In my project, a crucial aspect of data transformation involved creating unique identifiers for `OutcomeID`, `MedicationID`, and `AdverseEventID`. These identifiers were essential for effectively linking various tables in my relational database, ensuring that data relationships were maintained and could be queried efficiently.

Generating `MedicationID`:
I transformed the original data by consolidating various opioid-related attributes into a more manageable format. Each distinct medication was assigned a unique identifier (`MedicationID`), such as 'Med_1', 'Med_2', etc. This ID system allowed me to simplify the database schema by replacing multiple binary columns (indicating the presence or absence of each opioid) with a single column linking patients to their medications. Here’s how I handled this transformation:



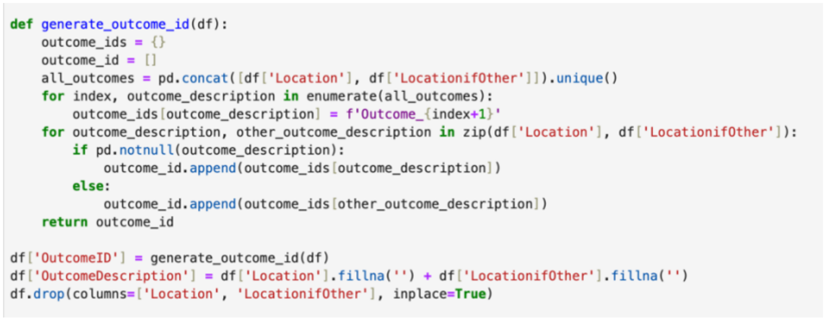
```python
def generate_medication_names(row):
    medication_names = []
    for opioid, medication_name in opioids_mapping.items():
        if row.get(opioid, 'N') == 'Y':
            medication_names.append(medication_name)
    return ','.join(medication_names) if medication_names else 'None'

df['MedicationName'] = df.apply(generate_medication_names, axis=1)

def combine_opioids(row):
    opioids = ['Heroin', 'Cocaine', 'Fentanyl', 'FentanylAnalogue', 'Oxycodone',
               'Oxymorphone', 'Ethanol', 'Hydrocodone', 'Benzodiazepine', 'Methadone',
               'Amphet', 'Tramad', 'Morphine_NotHeroin', 'Hydromorphone', 'Other',
               'OpiateNOS', 'AnyOpioid']
    medication_id = []
    for index, opioid in enumerate(opioids):
        if row[opioid] == 'Y':
            medication_id.append(f'Med_{index+1}')
    if medication_id:
        return ','.join(medication_id)
    else:
        return 'None'
```

Generating `AdverseEventID`:
For adverse events, I created a unique identifier for each distinct event description from the `COD` (Cause of Death) column in the dataset. This transformation was essential for normalizing the data and making it easier to link adverse events to patients and outcomes. By mapping each unique cause of death to an `AdverseEventID`, I enabled more efficient querying and aggregation in the database. Here's how I generated these IDs:



```python
def generate_adverse_event_id(df):
    event_ids = {}
    adverse_event_id = []
    for index, event_description in enumerate(df['COD'].unique()):
        event_id = f'Event_{index+1}'
        event_ids[event_description] = event_id
    for event_description in df['COD']:
        adverse_event_id.append(event_ids[event_description])
    return adverse_event_id

df['AdverseEventID'] = generate_adverse_event_id(df)
```

Generating `OutcomeID`:
Similarly, `OutcomeID` was generated to uniquely identify each combination of patient outcomes based on their location (hospital, home, etc.) and any other descriptors provided in the dataset. This allowed me to consolidate various descriptive attributes into a single identifier, making it much easier to manage relationships between adverse events and their outcomes in the database.



```python
def generate_outcome_id(df):
    outcome_ids = {}
    outcome_id = []
    all_outcomes = pd.concat([df['Location'], df['LocationifOther']]).unique()
    for index, outcome_description in enumerate(all_outcomes):
        outcome_ids[outcome_description] = f'Outcome_{index+1}'
    for outcome_description, other_outcome_description in zip(df['Location'], df['LocationifOther']):
        if pd.notnull(outcome_description):
            outcome_id.append(outcome_ids[outcome_description])
        else:
            outcome_id.append(outcome_ids[other_outcome_description])
    return outcome_id

df['OutcomeID'] = generate_outcome_id(df)
df['OutcomeDescription'] = df['Location'].fillna('') + df['LocationifOther'].fillna('')
df.drop(columns=['Location', 'LocationifOther'], inplace=True)
```

The rationale for creating these IDs was twofold: firstly, to ensure data integrity by preventing duplicate entries and inconsistencies across the database, and secondly, to improve query performance. With these IDs, I could more efficiently execute joins and aggregations across tables, enhancing the database's overall functionality and responsiveness when analyzing complex relationships in my data. This transformation was integral to maintaining a well-structured and efficient relational database environment, which is key to robust data analysis and reporting.

DATA VISUALISATION:

REQUIRED QUESTION:

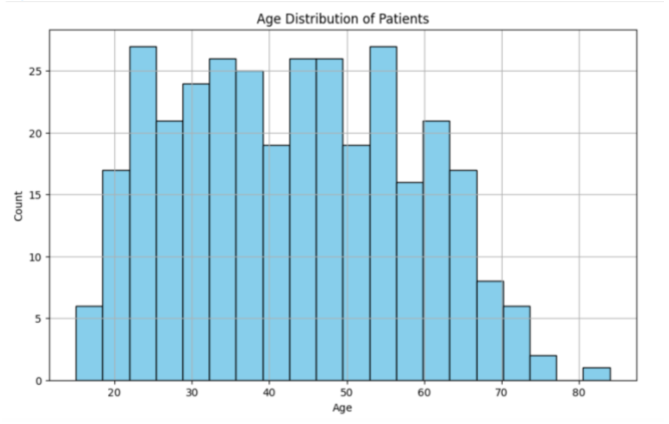
Detail your approach to data visualization and provide corresponding SQL queries.
In my project, I employed a structured approach to data visualization using Python's `matplotlib`, `seaborn`, and `folium` libraries to craft a comprehensive visual analysis of the dataset. Each visualization was carefully chosen to highlight specific aspects of the data, helping to draw meaningful insights and conclusions.

The codes for each visualization created and the corresponding SQL queries used to extract the necessary data:

```python
import matplotlib.pyplot as plt

ages = [float(row[0]) for row in demographic_results]
genders = [row[1] for row in demographic_results]
races = [row[2] for row in demographic_results]
counts = [int(row[3]) for row in demographic_results]
```

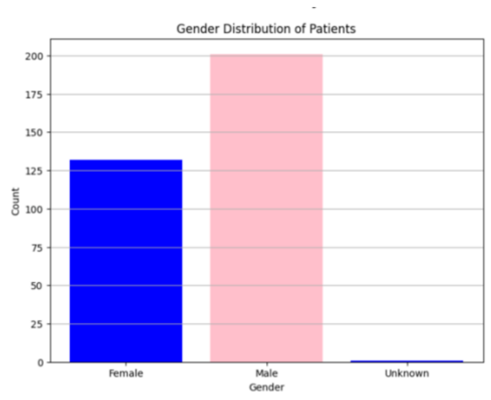
1. Age Distribution Histogram:
This visualization provided a clear view of the age distribution among the patients, highlighting which age groups are most commonly affected.

```python
plt.figure(figsize=(10, 6))
plt.hist(ages, bins=20, color='skyblue', edgecolor='black')
plt.title('Age Distribution of Patients')
plt.xlabel('Age')
plt.ylabel('Count')
plt.grid(True)
plt.show()
```



2. Gender Distribution Bar Chart:
A bar chart to show the gender distribution among patients, useful for identifying any gender-specific trends in the data.

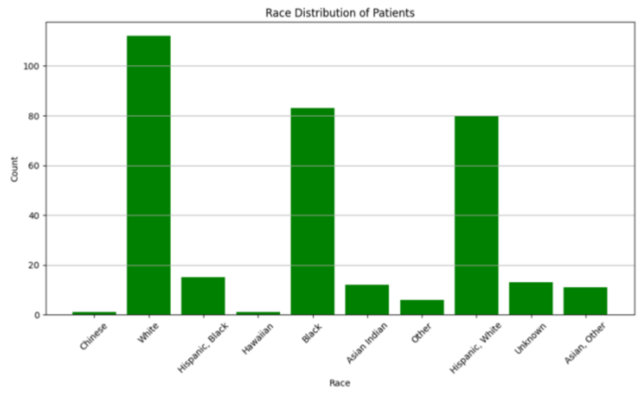
```python
plt.figure(figsize=(8, 6))
gender_counts = {gender: genders.count(gender) for gender in set(genders)}
plt.bar(gender_counts.keys(), gender_counts.values(), color=['blue', 'pink'])
plt.title('Gender Distribution of Patients')
plt.xlabel('Gender')
plt.ylabel('Count')
plt.grid(axis='y')
plt.show()
```



3. Race Distribution Bar Chart:

This bar chart compares the number of patients across different racial groups, essential for understanding racial disparities in drug-related incidents.

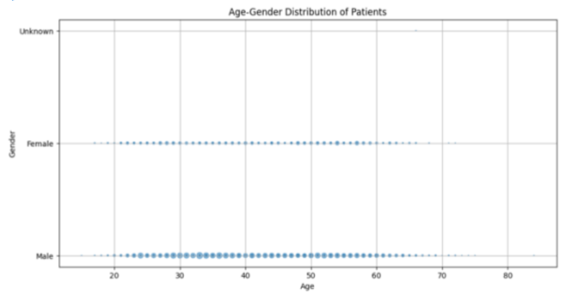
```
plt.figure(figsize=(12, 6))
race_counts = {race: races.count(race) for race in set(races)}
plt.bar(race_counts.keys(), race_counts.values(), color='green')
plt.title('Race Distribution of Patients')
plt.xlabel('Race')
plt.ylabel('Count')
plt.xticks(rotation=45)
plt.grid(axis='y')
plt.show()
```



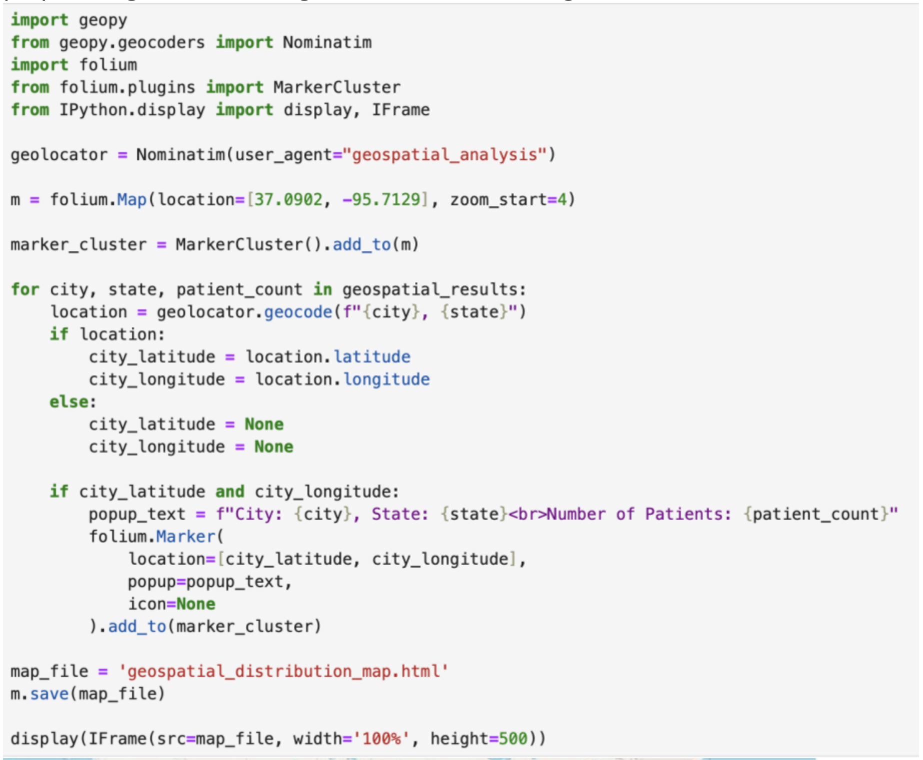
4. Age-Gender Distribution Scatter Plot:

A scatter plot to explore the relationship between age and gender among the patients, with the size of each point representing the number of patients.

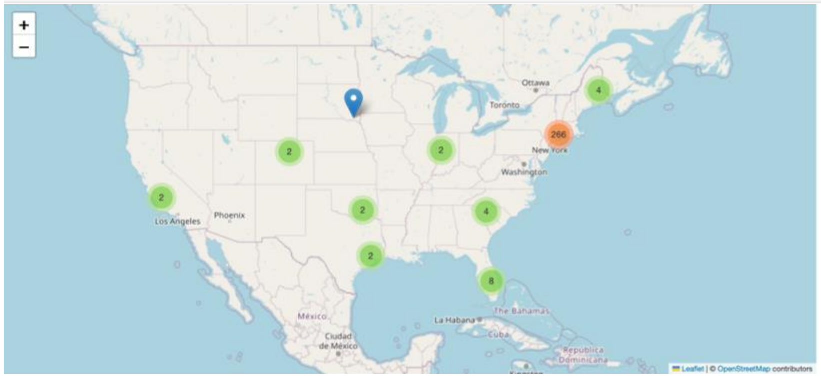
```
plt.figure(figsize=(12, 6))
plt.scatter(ages, genders, s=counts, alpha=0.5)
plt.title('Age-Gender Distribution of Patients')
plt.xlabel('Age')
plt.ylabel('Gender')
plt.grid(True)
plt.show()
```



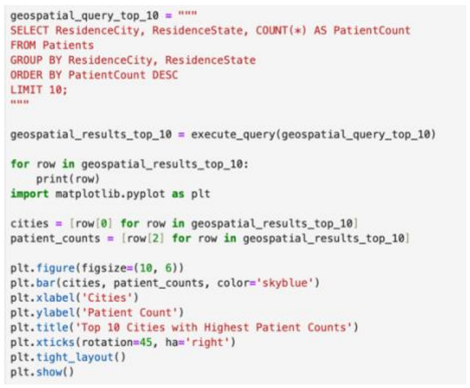
5. Geospatial Distribution Interactive Map:
An interactive map created using `folium` to visualize the geographical distribution of patients, pinpointing areas with higher incidences of drug-related issues.



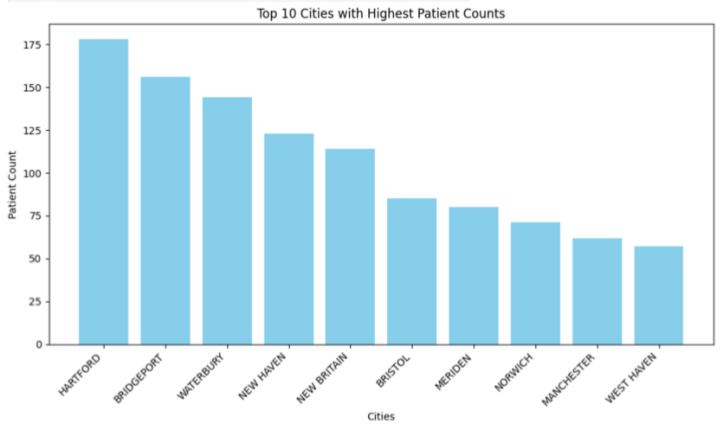
```python
import geopy
from geopy.geocoders import Nominatim
import folium
from folium.plugins import MarkerCluster
from IPython.display import display, IFrame

geolocator = Nominatim(user_agent="geospatial_analysis")

m = folium.Map(location=[37.0902, -95.7129], zoom_start=4)

marker_cluster = MarkerCluster().add_to(m)

for city, state, patient_count in geospatial_results:
    location = geolocator.geocode(f"{city}, {state}")
    if location:
        city_latitude = location.latitude
        city_longitude = location.longitude
    else:
        city_latitude = None
        city_longitude = None

    if city_latitude and city_longitude:
        popup_text = f"City: {city}, State: {state}<br>Number of Patients: {patient_count}"
        folium.Marker(
            location=[city_latitude, city_longitude],
            popup=popup_text,
            icon=None
        ).add_to(marker_cluster)

map_file = 'geospatial_distribution_map.html'
m.save(map_file)

display(IFrame(src=map_file, width='100%', height=500))
```



6. Top 10 Cities with Highest Patient Counts Bar Chart:
This bar chart ranks the top 10 cities by the number of patients, providing a clear visual representation of geographic areas with the highest drug-related issues.

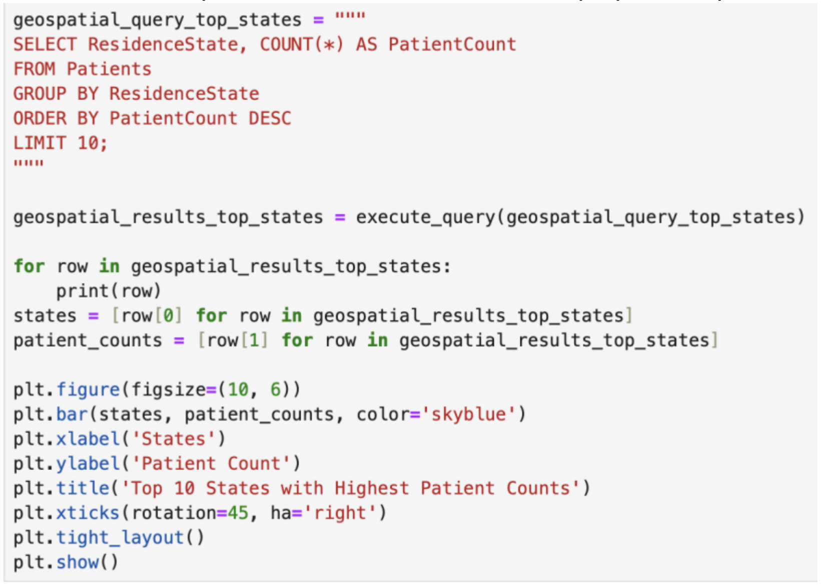


```
geospatial_query_top_10 = """
SELECT ResidenceCity, ResidenceState, COUNT(*) AS PatientCount
FROM Patients
GROUP BY ResidenceCity, ResidenceState
ORDER BY PatientCount DESC
LIMIT 10;
"""

geospatial_results_top_10 = execute_query(geospatial_query_top_10)

for row in geospatial_results_top_10:
    print(row)
import matplotlib.pyplot as plt

cities = [row[0] for row in geospatial_results_top_10]
patient_counts = [row[2] for row in geospatial_results_top_10]

plt.figure(figsize=(10, 6))
plt.bar(cities, patient_counts, color='skyblue')
plt.xlabel('Cities')
plt.ylabel('Patient Count')
plt.title('Top 10 Cities with Highest Patient Counts')
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()
```

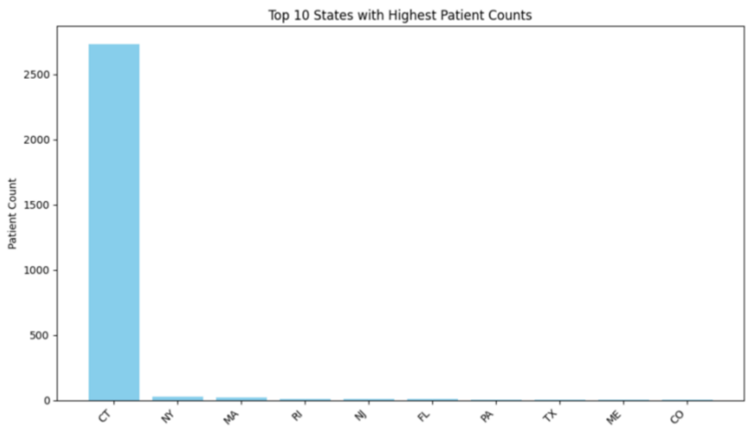


7. Top 10 States with Highest Patient Counts Bar Chart:

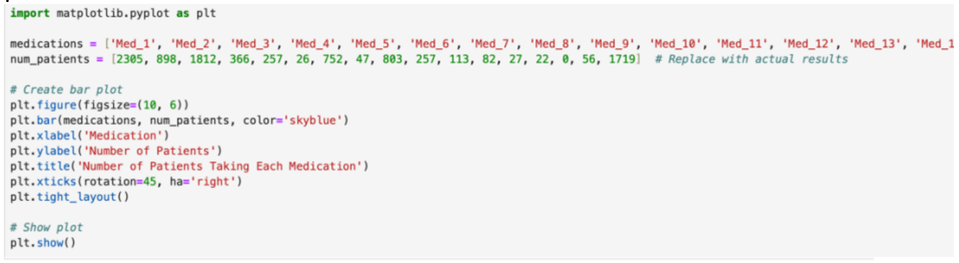
Similar to the city visualization, this bar chart displays the top 10 states by the number of patients.



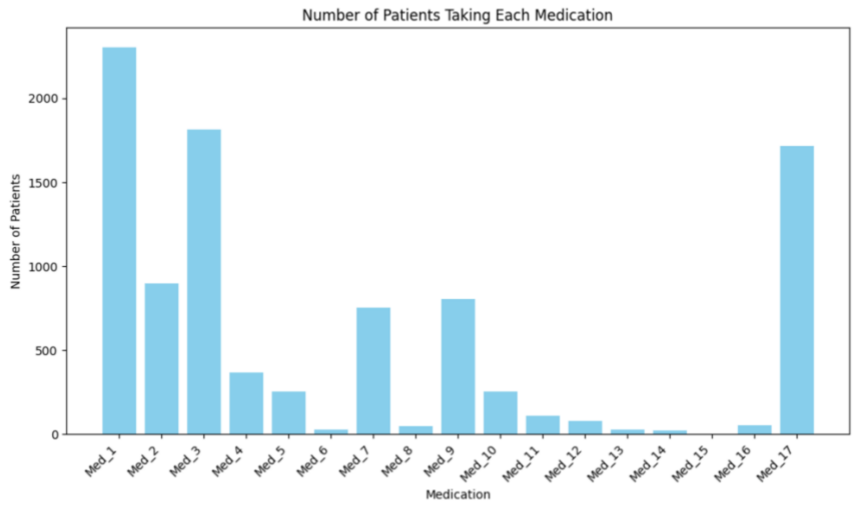
```
geospatial_query_top_states = """
SELECT ResidenceState, COUNT(*) AS PatientCount
FROM Patients
GROUP BY ResidenceState
ORDER BY PatientCount DESC
LIMIT 10;
"""

geospatial_results_top_states = execute_query(geospatial_query_top_states)

for row in geospatial_results_top_states:
    print(row)
states = [row[0] for row in geospatial_results_top_states]
patient_counts = [row[1] for row in geospatial_results_top_states]

plt.figure(figsize=(10, 6))
plt.bar(states, patient_counts, color='skyblue')
plt.xlabel('States')
plt.ylabel('Patient Count')
plt.title('Top 10 States with Highest Patient Counts')
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()
```



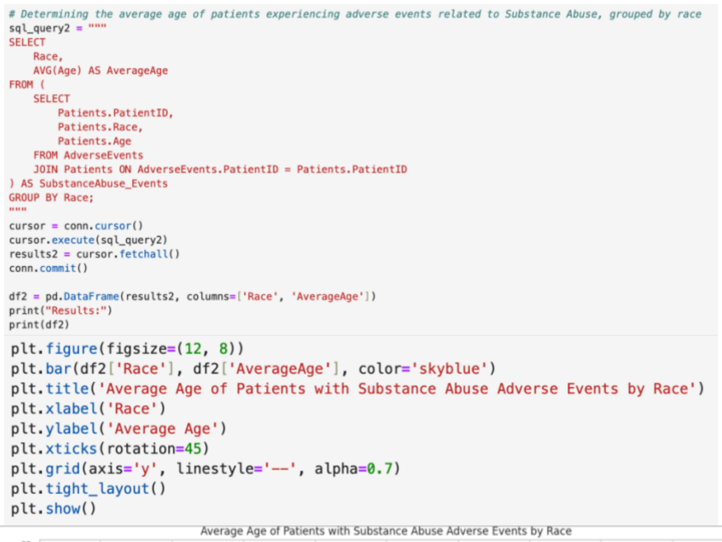
8. Medication Frequency Bar Chart:
This bar chart shows the frequency of each medication used by patients, highlighting the most prevalent medications.



```python
import matplotlib.pyplot as plt

medications = ['Med_1', 'Med_2', 'Med_3', 'Med_4', 'Med_5', 'Med_6', 'Med_7', 'Med_8', 'Med_9', 'Med_10', 'Med_11', 'Med_12', 'Med_13', 'Med_1
num_patients = [2305, 898, 1812, 366, 257, 26, 752, 47, 803, 257, 113, 82, 27, 22, 0, 56, 1719]  # Replace with actual results

# Create bar plot
plt.figure(figsize=(10, 6))
plt.bar(medications, num_patients, color='skyblue')
plt.xlabel('Medication')
plt.ylabel('Number of Patients')
plt.title('Number of Patients Taking Each Medication')
plt.xticks(rotation=45, ha='right')
plt.tight_layout()

# Show plot
plt.show()
```

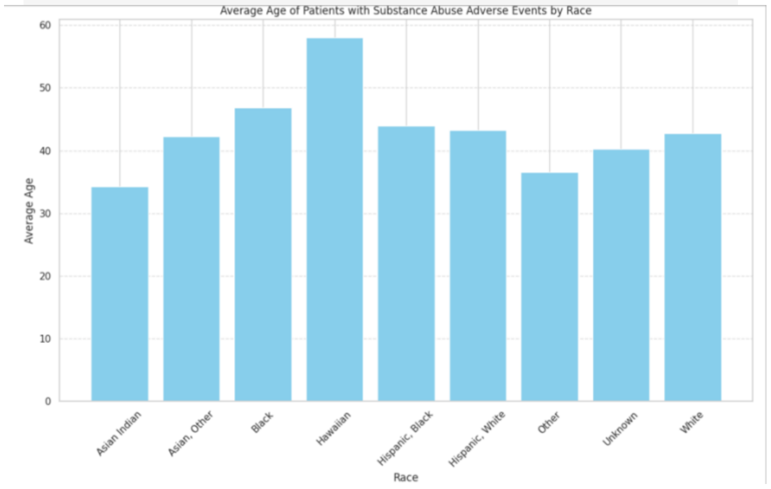


9. Determining the Average Age of Patients Experiencing Adverse Events Related to Substance Abuse, Grouped by Race:
This visualization provided insights into the average age of patients who have experienced adverse events related to substance abuse, broken down by race. It helps identify if certain racial groups are more susceptible at different ages.



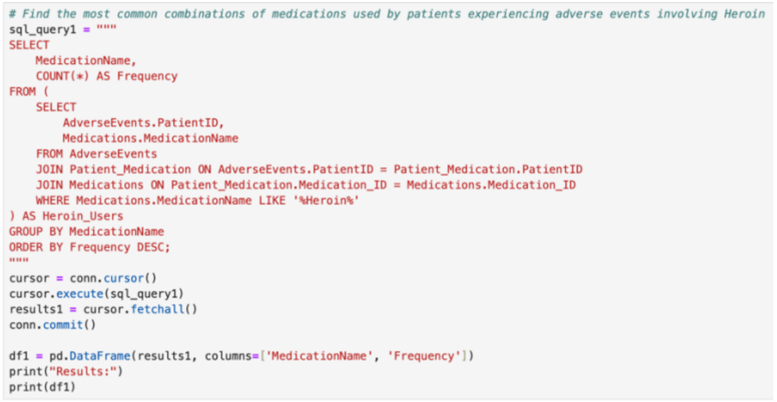
```
# Determining the average age of patients experiencing adverse events related to Substance Abuse, grouped by race
sql_query2 = """
SELECT
    Race,
    AVG(Age) AS AverageAge
FROM (
    SELECT
        Patients.PatientID,
        Patients.Race,
        Patients.Age
    FROM AdverseEvents
    JOIN Patients ON AdverseEvents.PatientID = Patients.PatientID
) AS SubstanceAbuse_Events
GROUP BY Race;
"""
cursor = conn.cursor()
cursor.execute(sql_query2)
results2 = cursor.fetchall()
conn.commit()

df2 = pd.DataFrame(results2, columns=['Race', 'AverageAge'])
print("Results:")
print(df2)
```

```
plt.figure(figsize=(12, 8))
plt.bar(df2['Race'], df2['AverageAge'], color='skyblue')
plt.title('Average Age of Patients with Substance Abuse Adverse Events by Race')
plt.xlabel('Race')
plt.ylabel('Average Age')
plt.xticks(rotation=45)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```



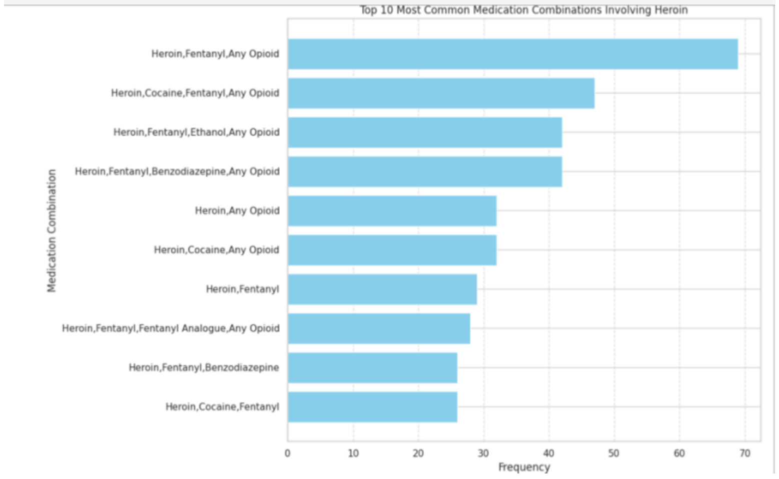
10. Most Common Medication Combinations Involving Heroin:

This analysis was aimed at identifying other medications that are commonly used in conjunction with



```
# Find the most common combinations of medications used by patients experiencing adverse events involving Heroin
sql_query1 = """
SELECT
    MedicationName,
    COUNT(*) AS Frequency
FROM (
    SELECT
        AdverseEvents.PatientID,
        Medications.MedicationName
    FROM AdverseEvents
    JOIN Patient_Medication ON AdverseEvents.PatientID = Patient_Medication.PatientID
    JOIN Medications ON Patient_Medication.Medication_ID = Medications.Medication_ID
    WHERE Medications.MedicationName LIKE '%Heroin%'
) AS Heroin_Users
GROUP BY MedicationName
ORDER BY Frequency DESC;
"""
cursor = conn.cursor()
cursor.execute(sql_query1)
results1 = cursor.fetchall()
conn.commit()

df1 = pd.DataFrame(results1, columns=['MedicationName', 'Frequency'])
print("Results:")
print(df1)
```

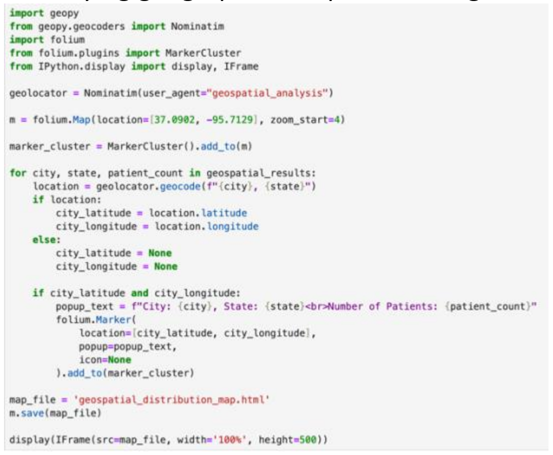
```python
top_n = 10
top_combinations = df1.head(top_n)

plt.figure(figsize=(12, 8))
plt.barh(top_combinations['MedicationName'], top_combinations['Frequency'], color='skyblue')
plt.title('Top 10 Most Common Medication Combinations Involving Heroin')
plt.xlabel('Frequency')
plt.ylabel('Medication Combination')
plt.gca().invert_yaxis()
plt.grid(axis='x', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```



Including this visualization enriches the analysis by giving healthcare providers and policymakers more detailed information on drug use patterns, particularly in the context of opioid abuse, which is a significant area of concern in public health.
Each of these visualizations was designed to provide specific insights into different aspects of the dataset, from demographic distributions and medication usage to geospatial patterns. This multifaceted approach ensured that stakeholders could gain a comprehensive understanding of the data and the underlying trends.

ELECTIVE QUESTIONS:

Demonstrate how you transformed a complex dataset into a meaningful visualization, including the underlying SQL query.
For this, I'll detail how I visualized the geospatial distribution of patients, which transformed a complex dataset of patient locations into an insightful and interactive map. This visualization was key to identifying geographic hotspots for drug-related incidents.



```python
import geopy
from geopy.geocoders import Nominatim
import folium
from folium.plugins import MarkerCluster
from IPython.display import display, IFrame

geolocator = Nominatim(user_agent="geospatial_analysis")

m = folium.Map(location=[37.0902, -95.7129], zoom_start=4)

marker_cluster = MarkerCluster().add_to(m)

for city, state, patient_count in geospatial_results:
    location = geolocator.geocode(f"{city}, {state}")
    if location:
        city_latitude = location.latitude
        city_longitude = location.longitude
    else:
        city_latitude = None
        city_longitude = None

    if city_latitude and city_longitude:
        popup_text = f"City: {city}, State: {state}<br>Number of Patients: {patient_count}"
        folium.Marker(
            location=[city_latitude, city_longitude],
            popup=popup_text,
            icon=None
        ).add_to(marker_cluster)

map_file = 'geospatial_distribution_map.html'
m.save(map_file)

display(IFrame(src=map_file, width='100%', height=500))
```

The interactive map uses `folium` to plot each city as a marker, with popup texts displaying detailed counts, enhancing the user's ability to explore data spatially. This transformation allows stakeholders to visualize and understand the geographic distribution of the patient data, which is crucial for deploying targeted public health interventions.

Discuss any challenges faced while visualizing large datasets and how you overcame them.
Visualizing large datasets often involves handling performance issues due to the sheer volume of data, which can slow down processing and make interactive visualizations less responsive.
Challenge:
One major challenge was the time it took to load and render the interactive maps when using large geospatial data sets. The maps were initially sluggish, impacting the user experience.
Solution:
To address this, I optimized the dataset by reducing the precision of the geographic coordinates and aggregating data at higher geographic levels (e.g., state or region instead of city) when appropriate. Additionally, I implemented marker clustering in `folium`, which groups nearby markers into a single cluster at various zoom levels. This not only improved the map's performance but also enhanced its usability by decluttering the visual presentation.

```python
marker_cluster = MarkerCluster().add_to(m)
```

By using clustering, the map loads fewer elements initially, speeding up the response times and improving user interaction. This solution was crucial in maintaining a balance between detailed geographical insights and performance, ensuring that the visualization remained a practical tool for analysis and decision-making.

INTEGRATING FEEDBACK:
In my presentation, feedback highlighted that my database tables were not in Third Normal Form (3NF). To address this, I introduced a new table, `PatientMedication`, in my midterm report, designating `PatientID` and `MedicationID` as composite primary keys to better decompose the data structure and ensure normalization. Additionally, during a consultation in the professor's office hours, it was suggested that mapping the columns for drug/medication usage directly in the database might complicate the schema unnecessarily. The professor advised simplifying the approach by reducing the direct mapping of these columns, potentially streamlining data management and enhancing the database's efficiency.