Data and Programming II: Final Project

Charisma Lambert & Lizzy Diaz

2024-12-07

Contributions

Lizzy Diaz	Charisma Lambert
Github: lizzydiaz	**Github**: charisml
Data cleaning	Data Cleaning
- Removal of columns not needed	- Initial exploration of data
- Remove columns without location data	- Adding Season column
- Creation of columns crash month, year, hour	- Creation of regression_df
- Merge original data on crash ID	- Take stratified sample (2021-2024)
- Random sample (2020-2024)	Visualization App
Crash maps	
- App and HTMLs	
Test apps (see Appendix III)	

Transportation and Public Safety

We examined vehicle crash data in order to answer the following questions: What is the state of traffic safety in Chicago? What factors influence vehicle crashes? These research questions came out of personal interest, and simply by observing the city around us. We recognize that traffic safety is a pervasive problem in Chicago, and analyzing this data can help to evaluate the scope of the problem and strategize targeted investments into improving the most affected areas.

Data and Methodology

We explored three datasets from the Chicago Data Portal:

1. Traffic Crashes - Crashes: Contains data on traffic crashes within city limits of Chicago and the reporting Chicago Police Department jurisdiction.
2. Traffic Crashes - People: Contains data on the people involved in traffic crashes and their level of injury. Involvement is defined as being an occupant in the vehicle at time of crash or the other party in the crash (pedestrian, cyclist, etc.).

From the information provided about the dataset and what we could decipher from cleaning, one accident can have multiple instances in all of the datasets. For example, look at the visual below. If there was a vehicular car crash involving two cars, where there were two and three people in the respective vehicles, the people dataset would have five instances, one for each person involved, and the crashes dataset might have 1 or 2, depending on if they are reporting the crash twice.

Our knowledge of how the different datasets were reporting information greatly informed how we cleaned the data. We were primarily interested in the types of people involved (vehicle drivers, passengers, pedestrians, cyclists, etc.), conditions that caused the crash (time of day, weather, variables describing the driver, etc.), and variables describing the severity of the crash (whether an airbag was deployed, worst injury sustained, etc.).

We combined 'Traffic Crashes - People' and 'Traffic Crashes - Crashes', and merged on Crash ID. We also used a geojson file of the Chicago ward boundaries to add a layer of spatial information to the final map. We kept observations from 2020-2024 for which there was reported location data. We removed observations where there was a >7-day discrepancy between the crash date given a unique crash ID across the two datasets.

We selected these years for a couple reasons: time effects and data limitations. Knowing the impact COVID-19 had on movement in 2020, we were curious to see how that pattern emerged on the map. There were 26,263 fewer unique crash IDs reported in 2020 compared to the average of 2018-2019. Additionally, it appeared that 2015-2016 had incomplete data or different reporting methodology. Finally, the dataset was quite large–the raw data file was about one million rows. Considering these limitations, we chose to only analyze data from 2020-2024 on the maps, and 2021-2024 for the statistical analysis.

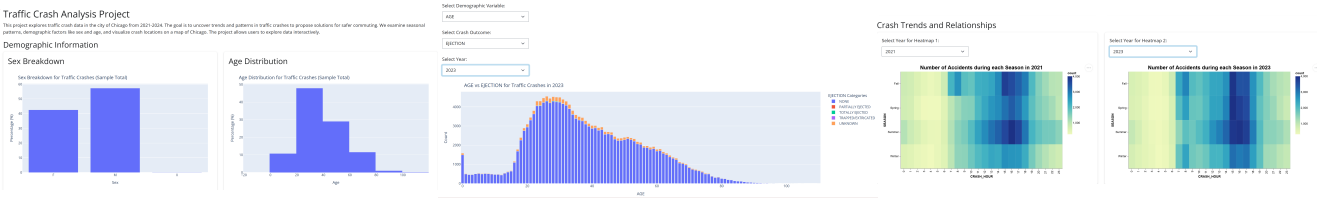
After cleaning, our dataset was too large for Github and to be processed by Shiny, so we had to truncate it even more to hold 1,000 random instances per year. The final Shiny app map contains a simple random sampling of 1,000 crash instances per year. For the statistical analysis app, the random sample was normalized to keep the same ratio of data from the original dataset in the sample (i.e., if the original dataset had 30% female and 70% male, our sample would have that same ratio). We maintained the original idea to visualize all crashes on a map, and achieved this by generating a simple HTML map of all crashes for a given year. We generated five maps, one for each year 2020-2025, which are available in our Google Drive data folder.

Approach

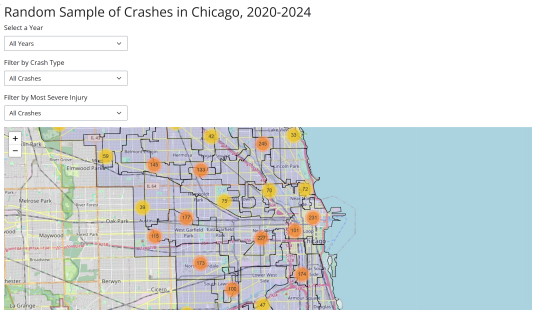
The final outcome of this project are two Shiny apps that allows users to take a "Choose Your Own Adventure" approach to understanding traffic crashes in Chicago. One allows the user to visualize the data and explore relationships between relevant variables (which we will refer to as the 'visualization app'), and the other displays a map where the user can filter observations by year, crash type, and severity ('mapp app').

Visualization App

- Static Visual of Demographic Information
- Static Heatmaps of Crash Occurences by Hour over the Seasons in the years 2021 - 2024
- Dynamic Variable Relationships Bar Chart: User Selects Variables and Year



Map App



- Dynamic Map of Chicago crashes: Users can zoom in, hover over data points, and explore different areas of the city.
- Filter: By year, crash type, and injury involved.
- The data folder also contains five accompanying HTML maps, one for each year (2020-2025).

In our analysis we examined the demographic data as independent variables and crash outcomes as dependent variables. The variables that users can choose from are: Age, Sex, Person_Type, Injury_Classification, Ejection, and Airbag_Deployed.

Results

From examining these variables, users can understand the following about Chicago crashes:

- Men are more likely to get into traffic accidents
- Time of day is important factor in crashes, with most occuring between the hours of 3-6pm
- The fewest accidents were in Spring 2021, otherwise each years' accident frequency is relatively the same
- More accidents occurring earlier in the day Fall and Summer
- There are a wide range of people involved, mostly 20-60 year-olds
- Vehicle crashes are are a clear problem throughout the entire city, with stability year-to-year
- Crashes appear to be concentrated in densley populated areas and high-traffic arterials

Future Research

The datasets contained far more data than we could have possibly examined within the scope of this project. There are numerous opportunities for a more complex spatial analysis of crashes and statistical analysis to predict factors most likely to affect a crash type, location, and severity. Further analysis could also explore where there is a disproportionate number of crashes to the amount of traffic that area receives, which would require even more traffic data than we currently have.

Finally, as we have touched on previously in this report, the data itself provided some limitations. There is room for improvement in the methods of reporting itself. For example, some columns suffered from noticeable missingness that appeared to be very relevant: for all pedestrian-involved crashes, 62% of observations are missing for whether it was a hit and run (a simple binary variable). Additionally, most reports are made at the police precinct after the crash has occurred. This means that there is often a lag between the crash date and the report date, which casts questions about some of the accuracy of the data.

Conclusions

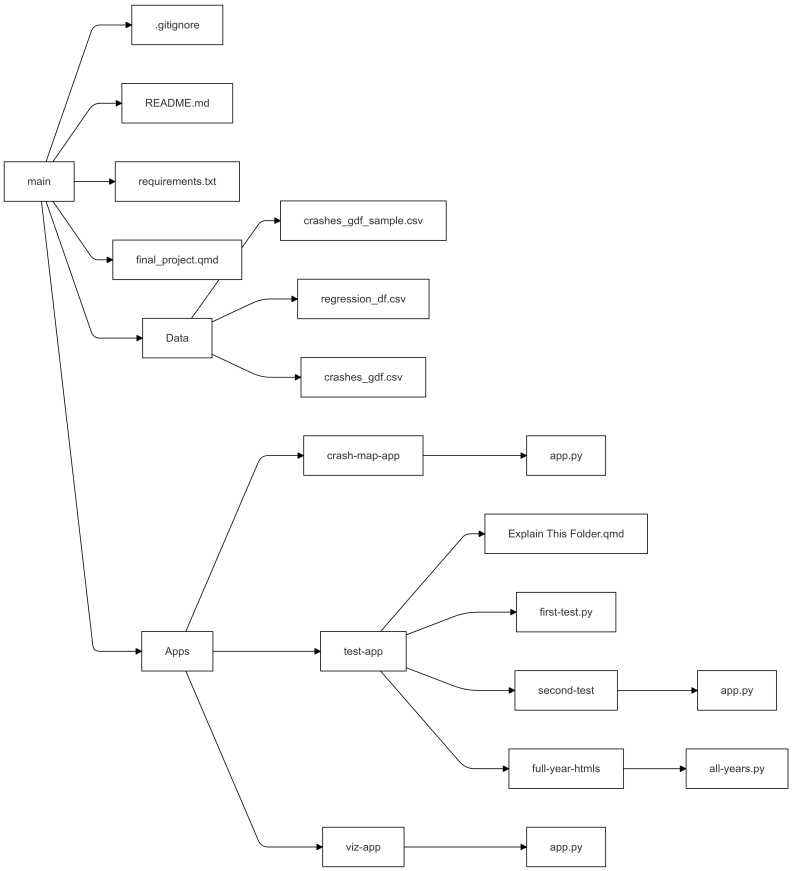
A simple glance at the size of the dataset provides perhaps the most important insight in this report: vehicle crashes are a tremendous problem in the City of Chicago. There is no clear pattern to their spatial distribution beyond high-traffic areas suffering from higher rates of crashes, meaning that the city overall could use long-term planning and investment into safer modes of transportation and infrastructure.

The distribution of crashes is stable from year-to-year with the exception of 2020. Vehicle traffic is increasing (1, 2), demonstrating that if safety improvements are being made, they are having some positive impact. The City should evaluate which of these improvements are having the strongest impact, and consider expanding those to other areas.

We hope these findings are just a foundation that other researchers can build upon to propose infrastructure and traffic control policies to make rush hour commuting in Chicago safer for drivers, pedestrians, cyclists, and more. The ward boundaries overlay on the maps provide an accountability measure, as alderpeople are responsible for directing infrastructure funding to roadways maintained by the Chicago Department of Transportation.

One major policy implication is the need to address the high frequency of crashes during rush hour (3-6pm). An intervention to consider is optimizing traffic light timings. Additionally, improving infrastructure like protected bike lanes and pedestrian-friendly designs can enhance safety for vulnerable road users. A city that does this well is Pittsburgh. Pittsburgh's use of the diagonal street crossings, known as the "Barnes Dance," optimizes crossing pathways for pedestrians and allows drivers to benefit from longer green lights, leading to less crashes due to the optimization of the traffic flow.

Appendix I: Structure of our Github Folder



Appendix II: Data Cleaning and Preparation

Install Packages for Document

```
import pandas as pd
import os
import csv
import altair as alt
import numpy as np
import warnings
import geopandas as gpd
from shapely import wkt
import matplotlib.pyplot as plt
import plotly.express as px
import random
warnings.filterwarnings("ignore")
```

Import raw data

```
raw_data_path = r"C:\Users\User\OneDrive - The University of Chicago\4_DAP-2\Final Project
 ↪  Data"
# raw_data_path = r"/Users/charismalambert/Downloads"

crashes = pd.read_csv(os.path.join(raw_data_path, "traffic_crashes_crashes.csv"))
people = pd.read_csv(os.path.join(raw_data_path, "traffic_crashes_people.csv"))
```

Keep data from 2016-present

```
dfs = [crashes, people]

# Convert the CRASH_DATE column to datetime and filter for 2016-present
for i in range(len(dfs)):
    dfs[i]['CRASH_DATE'] = pd.to_datetime(dfs[i]['CRASH_DATE'])
    dfs[i] = dfs[i][dfs[i]['CRASH_DATE'].dt.year >= 2016]
    dfs[i]['YEAR'] = dfs[i]['CRASH_DATE'].dt.year
```

```
# Function to count distinct CRASH_RECORD_ID per year
def count_distinct_crash_record_id(df):
    return df.groupby('YEAR')['CRASH_RECORD_ID'].nunique()
# Print distinct counts per year for each DataFrame
for i, df in enumerate(dfs, 1):
    print(f"DataFrame {i} distinct counts per year:\n", count_distinct_crash_record_id(df),
     ↪  "\n")
```

Remove observations where location data is unavailable

```
crashes_location = crashes.dropna(subset=['LOCATION'])
```

Remove columns we won't use

```
crashes_location = crashes_location.drop(columns=['TRAFFIC_CONTROL_DEVICE',
'DEVICE_CONDITION','INTERSECTION_RELATED_I','NOT_RIGHT_OF_WAY_I', 'BEAT_OF_OCCURRENCE',
'PHOTOS_TAKEN_I', 'STATEMENTS_TAKEN_I','WORK_ZONE_I', 'WORK_ZONE_TYPE',
'WORKERS_PRESENT_I', 'LANE_CNT', 'ALIGNMENT'])

people = people.drop(columns=['SEAT_NO', 'CITY', 'STATE', 'ZIPCODE',
'DRIVERS_LICENSE_STATE', 'DRIVERS_LICENSE_CLASS', 'SAFETY_EQUIPMENT',
'HOSPITAL','EMS_AGENCY', 'EMS_RUN_NO', 'BAC_RESULT', 'BAC_RESULT VALUE',
'CELL_PHONE_USE'])
```

Remove observations in people for which there is no corresponding 'crash_record_id' in crashes; do this by merging people and crashes_location

There are 161 observations where the dates for a given crash record ID do not match across the crashes and people datasets. If the mismatch in dates is greater than 7 days, I remove that observation from crashes_people.

```
crashes_people = pd.merge(people, crashes_location, on='CRASH_RECORD_ID', how='inner')

# Check merge worked by matching crash_date_x with crash_date_y
crashes_people['check'] = crashes_people['CRASH_DATE_x'] == crashes_people['CRASH_DATE_y']
crashes_people.shape[0] - crashes_people['check'].sum()

# View observations where dates don't match
mismatched_observations = crashes_people[crashes_people['check'] == False]
mismatched_observations['difference'] = mismatched_observations['CRASH_DATE_x'] -
 ↪  mismatched_observations['CRASH_DATE_y']
mismatched_observations['difference']= mismatched_observations['difference'].dt.days
```

```
# find any observations where the absolute value of the difference is greater than 7, identify
 ↪  those in the main dataset, and remove them
mismatched_observations['difference'] = mismatched_observations['difference'].abs()
mismatched_observations_over_7_days =
 ↪  mismatched_observations[mismatched_observations['difference'] > 7]

crashes_people =
 ↪  crashes_people[~crashes_people['CRASH_RECORD_ID'].isin(mismatched_observations_over_7_days['CRASH_RECO
```

Preparing to map

```
# Convert location column to geoseries
crashes_people['geometry'] = crashes_people['LOCATION'].apply(wkt.loads)
crashes_gdf = gpd.GeoDataFrame(crashes_people, geometry='geometry')

# reset CRS to IL
crashes_gdf.set_crs(epsg=4326, inplace=True)
crashes_gdf = crashes_gdf.to_crs(epsg=3435)

# Load the Chicago ward boundaries geojson
ward_boundaries = gpd.read_file(os.path.join(raw_data_path, 'ward_boundaries.geojson'))
ward_boundaries = ward_boundaries.to_crs(epsg=3435)


# Plot the ward boundaries and crash points
fig, ax = plt.subplots(figsize=(10, 10))
ward_boundaries.plot(ax=ax, color='lightgrey', edgecolor='black')
crashes_gdf.plot(ax=ax, color='red', markersize=1, alpha=0.7)
plt.show()
```

To make dataset manageable, keep only 2020-2024, and make year column. Drop unnecessary columns.

```
# Keep only crash_date_y (from crashes dataset)
crashes_gdf = crashes_gdf.drop(columns=['CRASH_DATE_x'])
crashes_gdf = crashes_gdf.rename(columns={'CRASH_DATE_y':'CRASH_DATE'})
crashes_gdf = crashes_gdf.drop(columns=['check'])

crashes_gdf['YEAR'] = crashes_gdf['CRASH_DATE'].dt.year

crashes_gdf = crashes_gdf[crashes_gdf['YEAR'] >= 2020]

# Add single address column
crashes_gdf['ADDRESS'] = crashes_gdf['STREET_NO'].astype(str) + " " +
 ↪  crashes_gdf['STREET_DIRECTION'] + " " + crashes_gdf['STREET_NAME']
crashes_gdf = crashes_gdf.drop(columns=['STREET_NO', 'STREET_DIRECTION',
'STREET_NAME'])
```

Save data to csv for app.

Create a new dataset to run map app - 1,000 random samples from each year

```
crashes_gdf_sample = crashes_gdf.groupby('YEAR').sample(n=1000, random_state=1)

# Combine address to single column and drop uncombined cols
```

```
crashes_gdf_sample['ADDRESS'] = crashes_gdf['STREET_NO'].astype(str) + " " +
 ↪  crashes_gdf['STREET_DIRECTION'] + " " + crashes_gdf['STREET_NAME']

crashes_gdf_sample = crashes_gdf_sample.drop(columns=['STREET_NO', 'STREET_DIRECTION',
'STREET_NAME'])
```

Preparing for Regression App

Create SEASON column

```
# add SEASON column to create heatmap of accidents by season
def fill_season(month):
    if month in [12, 1, 2]:
        return "Winter"
    elif month in [3, 4, 5]:
        return "Spring"
    elif month in [6, 7, 8]:
        return "Summer"
    else:
        return "Fall"

crashes_gdf["SEASON"] = crashes_gdf["CRASH_MONTH"].apply(fill_season)
```

Attempt Basic Heatmap before App integration

```
# Basic Code before app
tracking_by_season = crashes_gdf[crashes_gdf["YEAR"] == 2024].groupby(["SEASON",
 ↪  "CRASH_HOUR"]).size().reset_index(name = "count")

heatmap = alt.Chart(tracking_by_season).mark_rect().encode(
    x = "CRASH_HOUR:O",
    y = "SEASON:O",
    color = "count:Q",
).properties(
    width = 600,
    height = 300
)

heatmap
```

Code to add to the App:

```
# Dynamic for App:
# Include drop down for year (user modify) and crash_hour (constant)
# Year range: 2021 - 2024

def accidents_by_season(year):
    tracking_by_season = crashes_gdf[crashes_gdf["YEAR"] == year].groupby(["SEASON",
 ↪  "CRASH_HOUR"]).size().reset_index(name = "count")

    heatmap = alt.Chart(tracking_by_season).mark_rect().encode(
        x = "CRASH_HOUR:O",
        y = "SEASON:O",
        color = "count:Q",
```

```
    ).properties(
        title=f"Number of Accidents during each Season in {year}",
        width = 600,
        height = 300
    ).configure_title(
        fontSize = 18
    )

    heatmap.show()

accidents_by_season(2021)
```

Recode categorical column values to numbers

```
# convert IV column values to numerical values to make them regression friendly and store in
 ↪  new df
# recode SEX as M: 1, F: 0, X and Nan = 2:
crashes_gdf["SEX_R"] = crashes_gdf["SEX"].replace({"M": 1, "F": 0, "X": 2, None:
 ↪  -1}).astype("float64")

# recode SEASON as Winter: 1, Spring: 2, Summer: 3, Fall: 4
crashes_gdf["SEASON_R"] = crashes_gdf["SEASON"].replace({"Winter": 1, "Spring": 2, "Summer": 3,
 ↪  "Fall": 4}).astype("float64")


# recode PERSON_TYPE as Driver:0, Non-Driver(Passenger, Pedestrian,  Bicycle, Non-motor
 ↪  vehicle, Non-contact_vehicl): 1
crashes_gdf["PERSON_TYPE_R"] = crashes_gdf["PERSON_TYPE"].replace({
    'DRIVER':1,
    'PASSENGER': 2,
    'PEDESTRIAN': 3,
    'BICYCLE': 4,
    'NON-MOTOR VEHICLE': 5,
    'NON-CONTACT VEHICLE': 6
}).astype("float64")
```

```
# convert DV column values to numerical values to make them regression friendly

# recode AIRBAG_DEPLOYED as Deployed, Combo: 1, Deployed, Front: 1, Deployed, Side: 1, Deployed
 ↪  Other:1, Did Not Deploy: 0, Not Applicable: 0, Deployment Unknown: 0
crashes_gdf["AIRBAG_DEPLOYED_R"] = crashes_gdf["AIRBAG_DEPLOYED"].replace({
    'DEPLOYED, COMBINATION': 1,
    'DID NOT DEPLOY': 2,
    'NOT APPLICABLE': 3,
    'DEPLOYMENT UNKNOWN': 4,
    'DEPLOYED, FRONT': 5,
    'DEPLOYED, SIDE': 6,
    'DEPLOYED OTHER (KNEE, AIR, BELT, ETC.)': 7,
    None : -1
}).astype("float64")


# recode INJURY_CLASSIFICATION as Injured( 'NONINCAPACITATING INJURY': 1, 'REPORTED, NOT
 ↪  EVIDENT': 1,  'INCAPACITATING INJURY': 1, 'FATAL': 1) and Non-Injured ('NO INDICATION OF
 ↪  INJURY': 0, nan:0)
```

```
crashes_gdf["INJURY_CLASSIFICATION_R"] = crashes_gdf["INJURY_CLASSIFICATION"].replace({
    'NO INDICATION OF INJURY': 0,
    'NONINCAPACITATING INJURY': 1 ,
    'REPORTED, NOT EVIDENT': 2,
    'INCAPACITATING INJURY': 3 ,
    'FATAL': 4 ,
    None: -1
}).astype("float64")

# recode EJECTION as Ejected('TOTALLY EJECTED': 1, 'PARTIALLY EJECTED': 1) or Not-Ejeccted
 ↪  ('TRAPPED/EXTRICATED': 0, 'NONE': 0, 'UNKNOWN': 0,  nan: 0)
crashes_gdf["EJECTION_R"] = crashes_gdf["EJECTION"].replace({
    'TOTALLY EJECTED':  1,
    'TRAPPED/EXTRICATED' : 2,
    'PARTIALLY EJECTED': 3 ,
    'NONE': 0,
    'UNKNOWN': 0 ,
    None: - 1
}).astype("float64")


regression_df = crashes_gdf.dropna(subset = ["AGE"])
```

Save to new df

```
def reduce_data(data):
    # List of years in the dataset
    years = data["YEAR"].unique()

    # This will hold the reduced DataFrame
    reduced_df_list = []

    for year in years:
        # Filter the data for the current year
        year_data = data[data["YEAR"] == year]

        # Perform stratified sampling, keeping exactly 1000 rows for each year
        # Group by 'SEX_R' and 'PERSON_TYPE_R', and sample within each group
        stratified_sample = year_data.groupby(['SEX_R', 'PERSON_TYPE_R'],
 ↪  group_keys=False).apply(
            lambda x: x.sample(frac=1, random_state=42)
        )

        # Now sample 1000 rows from the stratified dataset
        if len(stratified_sample) > 1000:
            stratified_sample = stratified_sample.sample(n=1000, random_state=42)

        reduced_df_list.append(stratified_sample)

    # Concatenate all sampled dataframes into one
    reduced_df = pd.concat(reduced_df_list)

    return reduced_df

# Apply the function to reduce the size of the dataframe
```

```
regressions_df_reduced = reduce_data(regression_df)
```

Citation

Initially, I did a random sample of 1000 rows per year but I did not norm that my categorical variables were the same and when I ran the rest of my code, I noticed that SEX was only male, etc. I ran a query in ChatGPT on how to keep the proportion of categorical variables the same in my sample, as is in the original dataset. The query returned lines "stratified_sample".. and on.

Basic regressions before adding to App:

IV: AGE, SEX_R, SEASON_R, PERSON TYPE_R DV: AIRBAG_DEPLOYED_R, INJURY_CLASSIFICATION_R, EJECTION_R

```
def variable_relationships(IV, DV):
    if IV == "AGE":
        fig = px.box(regression_df,
            x = DV,
            y = IV,
            color = DV,
            title = f"{IV} vs {DV} for Traffic Crashes in 2024",
            labels = {IV: IV, DV: DV})
    else:
        fig = px.histogram(regression_df,
            x = IV,
            y = DV,
            barmode = 'group',
            title = f"{IV} vs {DV} for Traffic Crashes in 2024",
            labels = {IV: IV, DV: DV})
    fig.update_layout(legend_title_text = DV)
    fig.show()

variable_relationships("SEX", "EJECTION_R")
```

Code to add to the App:

```
# Drop down for IV, DV, YEAR
# IV's = Age, Sex, Season, Person_Type
# DV's = Injury_Classification, Airbag_Deployed, Ejection
# Year

# Based on what is chosen, produce regression stats that are user friendly

def variable_relationships(IV, DV, year):
    use_this_df = regression_df[regression_df["YEAR"] == year]
    if IV == "AGE":
        fig = px.box(use_this_df,
            x = DV,
            y = IV,
            color = DV,
            title = f"{IV} vs {DV} for Traffic Crashes in {year}",
            labels = {IV: IV, DV: DV})
    else:
        fig = px.histogram(use_this_df,
            x = IV,
            y = DV,
```

```
            barmode = 'group',
            title = f"{IV} vs {DV} for Traffic Crashes in 2024",
            labels = {IV: IV, DV: DV})
    fig.update_layout(legend_title_text = DV)
    fig.show()

variable_relationships("PERSON_TYPE", "EJECTION", 2023)
```

Appendix III: Explaining Test Apps

I used these apps to test and debug issues with getting a map to render inside the shiny app. For the final crashes app, I was trying to create an app that rendered an HTML file, saved it, then was opened inside the app. This kept returning a blank Shiny page, so I simplified the app to just be a map of Chicago's ward boundaries. I wanted to see if the problem was with the HTML, the saving, or Shiny. Here's an example of that code:

```
import shiny
from shiny import ui, render, Inputs, Outputs
import geopandas as gpd
import folium

# Load ward boundaries geojson
ward_boundaries = gpd.read_file(r'C:\Users\User\OneDrive - The University of
 ↪  Chicago\4_DAP-2\Final Project Data\ward_boundaries.geojson')
ward_boundaries = ward_boundaries.to_crs(epsg=3435)

# Define UI
app_ui = ui.page_fluid(
    ui.output_ui("map_ui")
)

def server(input, output, session):
    @output()
    @render.ui
    def map_ui():
        # Create map
        m = folium.Map(location=[41.882077, -87.627817], zoom_start=12)

        # Add ward boundaries to the map
        folium.GeoJson(
            ward_boundaries,
            name="Ward Boundaries",
            style_function=lambda x: {
                "fillColor": "blue",
                "color": "black",
                "weight": 1,
                "fillOpacity": 0.1,
            },
        ).add_to(m)

        # Save the map as an HTML file in the current working directory
        map_html = "simple_map.html"
        m.save(map_html)

        # Check if simple_map.html was saved correctly
        with open(map_html, 'r', encoding='utf-8') as file:
```

```
            map_content = file.read()

        # Return the map content within an iframe
        return ui.HTML(f'<iframe srcdoc="{map_content}" width="100%" height="600px"></iframe>')

app = shiny.App(app_ui, server)
```

I got the map I was expecting when I opened simple_map.html, but it wasn't appearing on the Shiny app. I first tried to debug issues with the file saving. I had trouble making it so that the HTML file was saving in the same folder as the app. I thought that the html content was not correctly loading into shiny because I got a 404 not found error in the browser. When I opened the HTML file directly, I was seeing the map I expected to see, but it was not appearing through Shiny.

Using Chatgpt, I explored a variety of ways to ensure that shiny was reading the correct file from the correct location. I further simplified the app to test these methods. Here are some examples, all of which were still producing the 404 Not Found error.

```
import shiny
from shiny import ui, render
import folium
import os

# Define UI
app_ui = ui.page_fluid(
    ui.output_ui("map_ui")  # Output UI for the map
)

def server(input, output, session):
    @output()
    @render.ui
    def map_ui():
        # Define the path to `www` relative to the script location
        script_dir = os.path.dirname(os.path.realpath(__file__))
        www_dir = os.path.join(script_dir, "www")
        os.makedirs(www_dir, exist_ok=True)

        # Save the map as an HTML file in the `www` directory under `test-app`
        map_html_path = os.path.join(www_dir, "simple_map.html")
        m = folium.Map(location=[41.882077, -87.627817], zoom_start=12)
        m.save(map_html_path)

        # Check if the file was saved correctly
        if not os.path.exists(map_html_path):
            raise FileNotFoundError(f"Map file not found: {map_html_path}")

        # Return iframe pointing to the file
        return ui.HTML('<iframe src="/simple_map.html" width="100%" height="600px"></iframe>')

app = shiny.App(app_ui, server)
```

```
import shiny
from shiny import ui, render
import folium
import os
from pathlib import Path
```

```
# Define UI
app_ui = ui.page_fluid(
    ui.output_ui("map_ui")  # Output UI for the map
)

def server(input, output, session):
    @output()
    @render.ui
    def map_ui():
        # Create the map
        m = folium.Map(location=[41.882077, -87.627817], zoom_start=12)

        # Add a simple marker for testing
        folium.Marker(
            location=[41.882077, -87.627817],
            popup="Test Marker",
            icon=folium.Icon(color='blue')
        ).add_to(m)

        # Save the map in the `www` directory
        app_dir = Path(__file__).parent  # Get the directory of this script
        www_dir = app_dir / "www"  # Create the path to the `www` folder
        os.makedirs(www_dir, exist_ok=True)
        map_html_path = www_dir / "simple_map.html"
        m.save(map_html_path)

        # Return iframe pointing to the file served via Shiny
        return ui.HTML('<iframe src="/static/simple_map.html" width="100%"
         ↪  height="600px"></iframe>')

# Create and run the app
app = shiny.App(app_ui, server, static_assets=str(Path(__file__).parent / "www"))
```

After ensuring that the files were being saved and read, and verifying that the HTML files looked accurate, I considered whether this may be a bug with Shiny itself. I found another person with these issues at https://github.com/posit-dev/py-shiny/issues/421 and by entering their sample solution into chatgpt to use that as a guide, I finally landed on the code in app.py:

```
from pathlib import Path
import shiny
from shiny import ui, render, reactive
import folium
import os

# Define paths
app_dir = Path(__file__).parent
www_dir = app_dir / "www"
www_dir.mkdir(exist_ok=True)  # Create `www` directory if it doesn't exist

# Define UI
app_ui = ui.page_fluid(
    ui.input_action_button("generate", "Generate Map"),  # Button to trigger map generation
    ui.output_ui("map_ui"),  # Output UI for the map
)
```

```
def server(input, output, session):
    @output
    @render.ui
    @reactive.event(input.generate)  # Render only when the button is clicked
    def map_ui():
        # Path for the HTML file
        map_html_path = www_dir / "simple_map.html"

        # Create and save the map
        m = folium.Map(location=[41.882077, -87.627817], zoom_start=12)
        folium.Marker(
            location=[41.882077, -87.627817],
            popup="Test Marker",
            icon=folium.Icon(color="blue"),
        ).add_to(m)
        m.save(map_html_path)

        # Debugging: Print paths and ensure everything is as expected
        print(f"App Directory: {app_dir}")
        print(f"www Directory: {www_dir}")
        print(f"Map HTML Path: {map_html_path}")
        print(f"Does file exist? {os.path.exists(map_html_path)}")

        # Return iframe pointing to the static asset
        return ui.HTML('<iframe src="simple_map.html" width="100%" height="600px"></iframe>')

# Define the app with static_assets pointing to `www_dir`
app = shiny.App(app_ui, server, static_assets=www_dir)
```

When I finally got my main app code to work, I noticed that it was producing and saving an HTML file, but that file was blank. However, the UI components of the app were present. I tested the code again with only 1000 samples, and everything worked perfectly! I think that because my app requires an HTML to be created then read in by Shiny, the HTML that was being created was too large, or something else going on that despite my best efforts I cannot figure out. I decided to move forward in two ways:

1. Create a new dataset with 1000 random observations from each year, 2020-2024. Use this dataset to run the Shiny app to show that all the desired UI features work.
2. In a separate .py file, located in /test-app/full-year-htmls, create an HTML file that contains a map with the full data for each year. In testing my code, I realized that if I created an HTML outside of Shiny–just by running a .py file–that it was able to produce a map with all of the 2024 data. This is how I came to the conclusion that perhaps Shiny was unable to handle the larger data. This folder contains 5 HTMLs, one for each year, that has a map with crash points and ward boundaries. Of course, it will not have any Shiny UI features, but I did want a way to visualize the complete data.