policing_project

November 30, 2022

Table of Contents

Stanford Open Policing Project.

[1]:
```python
# Load the necessary libraries
import pandas as pd
import json
import numpy as np
import matplotlib.pyplot as plt
import re
import seaborn as sns
```

1 Preparing data for analysis

Before I begin my analysis, it is critical that I first examine and clean the dataset, to make working with it a more efficient process. In this chapter, I will be fixing data types, handling missing values, and dropping columns and rows while learning about the Stanford Open Policing Project dataset.

1.1 Examining the dataset

Before beginning my analysis, it's important that I familiarize myself with the dataset. At this point, I'll read the dataset into pandas, examine the first few rows, and then count the number of missing values.

[2]:
```
# read the data
df = pd.read_csv('police.csv')
```

[3]:
```
# View the first few rows
df.head()
```

[3]:
```
  state   stop_date stop_time  county_name driver_gender driver_race  \
0    RI  2005-01-04     12:55          NaN             M       White
1    RI  2005-01-23     23:15          NaN             M       White
2    RI  2005-02-17     04:15          NaN             M       White
3    RI  2005-02-20     17:15          NaN             M       White
4    RI  2005-02-24     01:20          NaN             F       White

                    violation_raw  violation  search_conducted search_type  \
0  Equipment/Inspection Violation  Equipment             False         NaN
1                        Speeding   Speeding             False         NaN
2                        Speeding   Speeding             False         NaN
3                Call for Service      Other             False         NaN
4                        Speeding   Speeding             False         NaN

    stop_outcome is_arrested stop_duration  drugs_related_stop district
0       Citation       False      0-15 Min               False  Zone X4
1       Citation       False      0-15 Min               False  Zone K3
2       Citation       False      0-15 Min               False  Zone X4
3  Arrest Driver        True     16-30 Min               False  Zone X1
4       Citation       False      0-15 Min               False  Zone X3
```

[4]:
```
# Count the missing values
print(df.isnull().sum())
```

```
state                     0
stop_date                 0
stop_time                 0
county_name           91741
driver_gender          5205
driver_race            5202
```

```
violation_raw          5202
violation              5202
search_conducted          0
search_type           88434
stop_outcome           5202
is_arrested            5202
stop_duration          5202
drugs_related_stop        0
district                  0
dtype: int64
```

It is clear that most of the columns have some missing values.

1.2 Data Wrangling

1.2.1 Dropping column

Often, a DataFrame will contain columns that are not useful to your analysis. Such columns should be dropped from the DataFrame, to make it easier for you to focus on the remaining columns.

I'll drop the county_name column because it only contains missing values. Thus, these column can be dropped because it contain no useful information.

[5]:
```
# shape of the dataframe
df.shape
```

[5]:
```
(91741, 15)
```

[6]:
```
# drop the 'county_name'
df.drop(['county_name', 'state'], axis = 'columns', inplace = True)
```

[7]:
```
# Re-examine the shape of the data
df.shape
```

[7]:
```
(91741, 13)
```

1.2.2 Dropping rows

When you know that a specific column will be critical to your analysis, and only a small fraction of rows are missing a value in that column, it often makes sense to remove those rows from the dataset.

The driver_gender column will be critical to many of my analyses. Because only a small fraction of rows are missing driver_gender, I'll drop those rows from the dataset.

[8]:
```
# Drop all rows with missing values driver_gender column
df.dropna(subset = ['driver_gender'], inplace = True)
```

[9]:
```
# count the missing values
df.isnull().sum()
```

[9]:
```
stop_date                0
stop_time                0
driver_gender            0
driver_race              0
violation_raw            0
violation                0
search_conducted         0
search_type          83229
stop_outcome             0
is_arrested              0
stop_duration            0
drugs_related_stop       0
district                 0
dtype: int64
```

I dropped around 5,000 rows, which is a small fraction of the dataset.

1.2.3 Fixing a data type

[10]:
```python
# Examine the first five rows
df.head()
```

[10]:
```
    stop_date stop_time driver_gender driver_race  \
0  2005-01-04     12:55             M       White
1  2005-01-23     23:15             M       White
2  2005-02-17     04:15             M       White
3  2005-02-20     17:15             M       White
4  2005-02-24     01:20             F       White

                    violation_raw  violation  search_conducted search_type  \
0  Equipment/Inspection Violation  Equipment             False         NaN
1                        Speeding   Speeding             False         NaN
2                        Speeding   Speeding             False         NaN
3                Call for Service      Other             False         NaN
4                        Speeding   Speeding             False         NaN

    stop_outcome is_arrested stop_duration  drugs_related_stop district
0       Citation       False      0-15 Min               False  Zone X4
1       Citation       False      0-15 Min               False  Zone K3
2       Citation       False      0-15 Min               False  Zone X4
3  Arrest Driver        True     16-30 Min               False  Zone X1
4       Citation       False      0-15 Min               False  Zone X3
```

is_arrested column currently has the object data type. therefore, i'll change the data type to bool, which is the most suitable type for a column containing True and False values.

Fixing the data type will enable us to use mathematical operations on the is_arrested column that would not be possible otherwise.

[11]:
```
# Change the data type of 'is_arrested' to 'bool'
df['is_arrested'] = df.is_arrested.astype(bool)
```

[12]:
```
# Check the data type of 'is_arrested'
print(df.is_arrested.dtype)
```

```
bool
```

1.2.4 Combining object columns

Currently, the date and time of each traffic stop are stored in separate object columns: stop_date and stop_time.

To fix this, I'll combine these two columns into a single column, and then convert it to datetime format. This will enable convenient date-based attributes that I'll use later in the analysis.

[13]:
```
# Concatenate 'stop_date' and 'stop_time' (separated by a space)
combined = df.stop_date.str.cat(df.stop_time, sep=' ')

# Convert 'combined' to datetime format
df['stop_datetime'] = pd.to_datetime(combined)

# Examine the data types of the DataFrame
print(df.dtypes)
```

```
stop_date                     object
stop_time                     object
driver_gender                 object
driver_race                   object
violation_raw                 object
violation                     object
search_conducted                bool
search_type                   object
stop_outcome                  object
is_arrested                     bool
stop_duration                 object
drugs_related_stop              bool
district                      object
stop_datetime         datetime64[ns]
dtype: object
```

1.2.5 Setting the index

The last step that I'll take in this chapter is to set the stop_datetime column as the DataFrame's index. By replacing the default index with a DatetimeIndex, I'll make it easier to analyze the dataset by date and time, which will come in handy later in the analysis

[14]:
```
# set the stop_datetime as the index
df.set_index('stop_datetime', inplace = True)
```

```python
#examine the index
print(df.index)

# examine the columns
print(df.columns)
```

```
DatetimeIndex(['2005-01-04 12:55:00', '2005-01-23 23:15:00',
               '2005-02-17 04:15:00', '2005-02-20 17:15:00',
               '2005-02-24 01:20:00', '2005-03-14 10:00:00',
               '2005-03-29 21:55:00', '2005-04-04 21:25:00',
               '2005-07-14 11:20:00', '2005-07-14 19:55:00',
               …
               '2015-12-31 13:23:00', '2015-12-31 18:59:00',
               '2015-12-31 19:13:00', '2015-12-31 20:20:00',
               '2015-12-31 20:50:00', '2015-12-31 21:21:00',
               '2015-12-31 21:59:00', '2015-12-31 22:04:00',
               '2015-12-31 22:09:00', '2015-12-31 22:47:00'],
              dtype='datetime64[ns]', name='stop_datetime', length=86536,
freq=None)
Index(['stop_date', 'stop_time', 'driver_gender', 'driver_race',
       'violation_raw', 'violation', 'search_conducted', 'search_type',
       'stop_outcome', 'is_arrested', 'stop_duration', 'drugs_related_stop',
       'district'],
      dtype='object')
```

2 Exploring the relationship between gender and policing

Does the gender of a driver have an impact on police behavior during a traffic stop? In this chapter, I will explore that question while filtering, grouping, method chaining, Boolean math, string methods, and more!

2.1 Does gender affect who gets a ticket for speeding?

2.1.1 Examining traffic violations

Before comparing the violations being committed by each gender, I should examine the violations committed by all drivers to get a baseline understanding of the data.

In this exercise, I'll count the unique values in the violation column, and then separately express those counts as proportions.

[15]:
```python
# Count the unique values in 'violation'
print('unique_values:\n', df.violation.value_counts())
```

```
unique_values:
 Speeding               48423
Moving violation       16224
Equipment              10921
```

```
Other                     4409
Registration/plates       3703
Seat belt                 2856
Name: violation, dtype: int64
```

[16]:
```
# Express the counts as proportions
print('proportion:\n', df.violation.value_counts(normalize=True))
```

```
proportion:
 Speeding                0.559571
Moving violation       0.187483
Equipment              0.126202
Other                  0.050950
Registration/plates    0.042791
Seat belt              0.033004
Name: violation, dtype: float64
```

Interesting! More than half of all violations are for speeding, followed by other moving violations and equipment violations.

2.1.2 Comparing violations by gender

The question I am trying to answer is whether male and female drivers tend to commit different types of traffic violations.

In this exercise, I'll first create a DataFrame for each gender, and then analyze the violations in each DataFrame separately.

[17]:
```
# Create a DataFrame of female drivers
female = df[df.driver_gender== 'F']

# Create a DataFrame of male drivers
male = df[df.driver_gender== 'M']
```

[18]:
```
# Compute the violations by female drivers (as proportions)
print(female.violation.value_counts(normalize=True))
```

```
Speeding               0.658114
Moving violation       0.138218
Equipment              0.105199
Registration/plates    0.044418
Other                  0.029738
Seat belt              0.024312
Name: violation, dtype: float64
```

[19]:
```
# Compute the violations by male drivers (as proportions)
print(male.violation.value_counts(normalize=True))
```

```
Speeding               0.522243
Moving violation       0.206144
```

```
Equipment              0.134158
Other                  0.058985
Registration/plates    0.042175
Seat belt              0.036296
Name: violation, dtype: float64
```

About two-thirds of female traffic stops are for speeding, whereas stops of males are more balanced among the six categories. This doesn't mean that females speed more often than males, however, since we didn't take into account the number of stops or drivers.

2.2 Does gender affect who gets a ticket for overspeeding

Comparing speeding outcomes by gender

When a driver is pulled over for speeding, many people believe that gender has an impact on whether the driver will receive a ticket or a warning. Can I find evidence of this in the dataset?

First, I'll create two DataFrames of drivers who were stopped for speeding: one containing females and the other containing males.

Then, for each gender, I'll use the stop_outcome column to calculate what percentage of stops resulted in a "Citation" (meaning a ticket) versus a "Warning".

[20]:
```
# Create a DataFrame of female drivers stopped for speeding
female_and_speeding = df[(df.driver_gender == 'F') & (df.violation ==␣
↪'Speeding')]

# Create a DataFrame of male drivers stopped for speeding
male_and_speeding = df[(df.driver_gender == 'M') & (df.violation == 'Speeding')]
```

[21]:
```
# Compute the stop outcomes for female drivers (as proportions)
print(female_and_speeding.stop_outcome.value_counts(normalize=True))
```

```
Citation            0.952192
Warning             0.040074
Arrest Driver       0.005752
N/D                 0.000959
Arrest Passenger    0.000639
No Action           0.000383
Name: stop_outcome, dtype: float64
```

[22]:
```
# Compute the stop outcomes for male drivers (as proportions)
print(male_and_speeding.stop_outcome.value_counts(normalize=True))
```

```
Citation            0.944595
Warning             0.036184
Arrest Driver       0.015895
Arrest Passenger    0.001281
No Action           0.001068
```

```
N/D                  0.000976
Name: stop_outcome, dtype: float64
```

Interesting! The numbers are similar for males and females: about 95% of stops for speeding result in a ticket. Thus, the data fails to show that gender has an impact on who gets a ticket for speeding.

2.3 Does gender affect whose vehicle is searched?

2.3.1 Calculating the search rate

During a traffic stop, the police officer sometimes conducts a search of the vehicle. In the cell that follws, I'll calculate the percentage of all stops in the ri DataFrame that result in a vehicle search, also known as the search rate.

[23]:
```
# Check the data type of 'search_conducted'
print(df.search_conducted.dtype)
```

```
bool
```

[24]:
```
# Calculate the search rate by counting the values
print(df.search_conducted.value_counts(normalize = True))
```

```
False    0.961785
True     0.038215
Name: search_conducted, dtype: float64
```

[25]:
```
# Calculate the search rate by taking the mean
print(df.search_conducted.mean())
```

```
0.0382153092354627
```

Great! It looks like the search rate is about 3.8%. Next, you'll examine whether the search rate varies by driver gender.

2.3.2 Comparing search rates by gender

First, I'll filter the DataFrame by gender and calculate the search rate for each group separately. Then, I'll perform the same calculation for both genders at once using a .groupby()

[26]:
```
# Calculate the search rate for female drivers
print(df[df.driver_gender=='F'].search_conducted.mean())
```

```
0.019180617481282074
```

[27]:
```
# Calculate the search rate for male drivers
print(df[df.driver_gender=='M'].search_conducted.mean())
```

```
0.04542557598546892
```

[28]:
```
# Calculate the search rate for both groups simultaneously
print(df.groupby('driver_gender').search_conducted.mean())
```

```
driver_gender
F    0.019181
M    0.045426
Name: search_conducted, dtype: float64
```

Male drivers are searched more than twice as often as female drivers.

2.3.3 Adding a second factor to the analysis

Even though the search rate for males is much higher than for females, it's possible that the difference is mostly due to a second factor.

For example, you might hypothesize that the search rate varies by violation type, and the difference in search rate between males and females is because they tend to commit different violations.

You can test this hypothesis by examining the search rate for each combination of gender and violation. If the hypothesis was true, you would find that males and females are searched at about the same rate for each violation. Let's find out if that's the case!

[29]:
```python
# Calculate the search rate for each combination of gender and violation
print(df.groupby(['driver_gender', 'violation']).search_conducted.mean())
```

```
driver_gender  violation
F              Equipment              0.039984
               Moving violation       0.039257
               Other                  0.041018
               Registration/plates    0.054924
               Seat belt              0.017301
               Speeding               0.008309
M              Equipment              0.071496
               Moving violation       0.061524
               Other                  0.046191
               Registration/plates    0.108802
               Seat belt              0.035119
               Speeding               0.027885
Name: search_conducted, dtype: float64
```

[30]:
```python
# Reverse the ordering to group by violation before gender
print(df.groupby(['violation', 'driver_gender']).search_conducted.mean())
```

```
violation            driver_gender
Equipment            F                0.039984
                     M                0.071496
Moving violation     F                0.039257
                     M                0.061524
Other                F                0.041018
                     M                0.046191
Registration/plates  F                0.054924
                     M                0.108802
Seat belt            F                0.017301
```

```
                     M                 0.035119
Speeding             F                 0.008309
                     M                 0.027885
Name: search_conducted, dtype: float64
```

For all types of violations, the search rate is higher for males than for females, disproving our hypothesis.

2.4 Does gender affect who is frisked during a search

2.4.1 Counting protective frisks

During a vehicle search, the police officer may pat down the driver to check if they have a weapon. This is known as a "protective frisk."

In this point, I'll first check to see how many times "Protective Frisk" was the only search type. Then, I'll use a string method to locate all instances in which the driver was frisked.

[31]:
```python
# Count the 'search_type' values
print(df.search_type.value_counts())
```

```
Incident to Arrest                                          1290
Probable Cause                                               924
Inventory                                                    219
Reasonable Suspicion                                         214
Protective Frisk                                             164
Incident to Arrest,Inventory                                 123
Incident to Arrest,Probable Cause                            100
Probable Cause,Reasonable Suspicion                           54
Incident to Arrest,Inventory,Probable Cause                   35
Probable Cause,Protective Frisk                               35
Incident to Arrest,Protective Frisk                           33
Inventory,Probable Cause                                      25
Protective Frisk,Reasonable Suspicion                         19
Incident to Arrest,Inventory,Protective Frisk                 18
Incident to Arrest,Probable Cause,Protective Frisk            13
Inventory,Protective Frisk                                    12
Incident to Arrest,Reasonable Suspicion                        8
Probable Cause,Protective Frisk,Reasonable Suspicion           5
Incident to Arrest,Probable Cause,Reasonable Suspicion         5
Incident to Arrest,Inventory,Reasonable Suspicion              4
Incident to Arrest,Protective Frisk,Reasonable Suspicion       2
Inventory,Reasonable Suspicion                                 2
Inventory,Protective Frisk,Reasonable Suspicion                1
Inventory,Probable Cause,Reasonable Suspicion                  1
Inventory,Probable Cause,Protective Frisk                      1
Name: search_type, dtype: int64
```

[32]:
```
# Check if 'search_type' contains the string 'Protective Frisk'
df['frisk'] = df.search_type.str.contains('Protective Frisk', na=False)
```

[33]:
```
# Check the data type of 'frisk'
print(df.frisk.dtype)
```

```
bool
```

[34]:
```
# Take the sum of 'frisk'
print(df.frisk.sum())
```

```
303
```

It looks like there were 303 drivers who were frisked. Next, you'll examine whether gender affects who is frisked.

2.4.2 Comparing frisk rates by gender

I'll compare the rates at which female and male drivers are frisked during a search. Are males frisked more often than females, perhaps because police officers consider them to be higher risk?

Before doing any calculations, it's important to filter the DataFrame to only include the relevant subset of data, namely stops in which a search was conducted.

[35]:
```
# Create a DataFrame of stops in which a search was conducted
searched = df[df.search_conducted == True]

# Calculate the overall frisk rate by taking the mean of 'frisk'
print('Mean: ', searched.frisk.mean())

# Calculate the frisk rate for each gender
print(searched.groupby('driver_gender')['frisk'].mean())
```

```
Mean:  0.09162382824312065
driver_gender
F    0.074561
M    0.094353
Name: frisk, dtype: float64
```

The frisk rate is higher for males than for females, though we can't conclude that this difference is caused by the driver's gender.

3 Visual exploratory data analysis

Are you more likely to get arrested at a certain time of day? Are drug-related stops on the rise? In this chapter, I will answer these and other questions by analyzing the dataset visually, since plots can help me to understand trends in a way that examining the raw data cannot.

3.1 Does time of day affect arrest rate?

3.1.1 Calculating the hourly arrest rate

When a police officer stops a driver, a small percentage of those stops ends in an arrest. This is known as the arrest rate. I'll find out whether the arrest rate varies by time of day.

First, I'll calculate the arrest rate across all stops in the df DataFrame. Then, I'll calculate the hourly arrest rate by using the hour attribute of the index. The hour ranges from 0 to 23, in which:

- 0 = midnight
- 12 = noon
- 23 = 11 PM

[36]:
```
# Calculate the overall arrest rate
print(df.is_arrested.mean())
```

```
0.0355690117407784
```

[37]:
```
# Save the hourly arrest rate
hourly_arrest_rate = df.groupby(df.index.hour).is_arrested.mean()

# Calculate the hourly arrest rate
print(hourly_arrest_rate)
```

```
stop_datetime
0     0.051431
1     0.064932
2     0.060798
3     0.060549
4     0.048000
5     0.042781
6     0.013813
7     0.013032
8     0.021854
9     0.025206
10    0.028213
11    0.028897
12    0.037399
13    0.030776
14    0.030605
15    0.030679
16    0.035281
17    0.040619
18    0.038204
19    0.032245
20    0.038107
21    0.064541
22    0.048666
23    0.047592
```

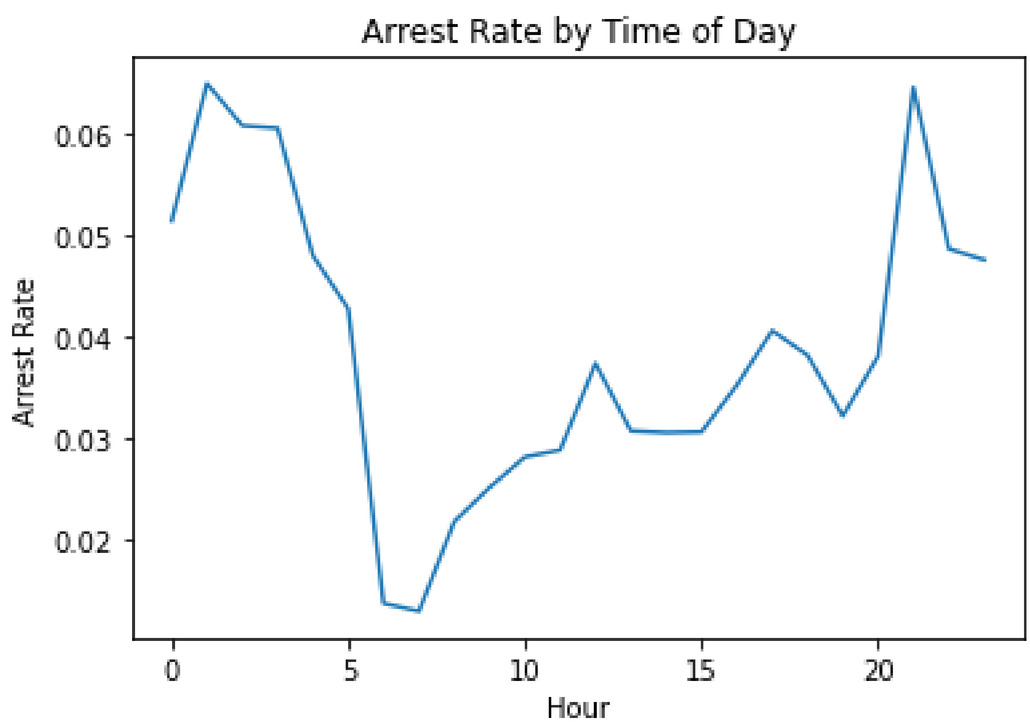
```
Name: is_arrested, dtype: float64
```

3.1.2 Plotting the hourly arrest rate

I'll create a line plot from the hourly_arrest_rate object. A line plot is appropriate in this case because I am showing how a quantity changes over time.

This plot should help me to spot some trends that may not have been obvious when examining the raw numbers!

[38]:
```
# Create a line plot of 'hourly_arrest_rate'

hourly_arrest_rate.plot()
# Add the xlabel, ylabel, and title
plt.xlabel('Hour')
plt.ylabel('Arrest Rate')
plt.title('Arrest Rate by Time of Day')

# Display the plot
plt.show()
```



The arrest rate has a significant spike overnight, and then dips in the early morning hours.

3.2 Are drug-related stops on the rise?

Plotting drug-related stops

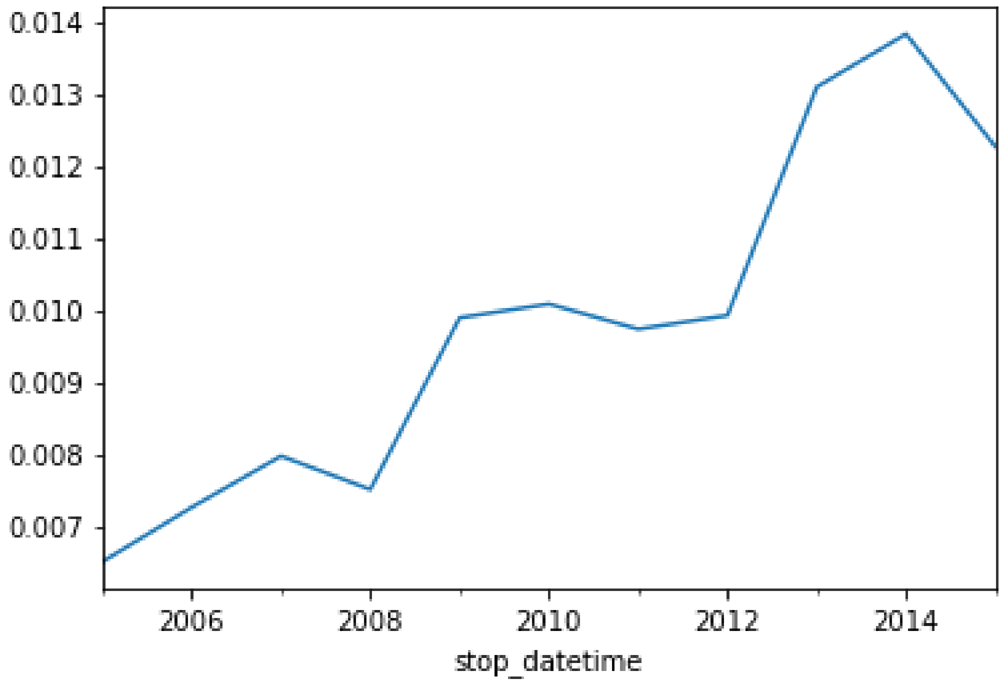
In a small portion of traffic stops, drugs are found in the vehicle during a search. In this case, I'll assess whether these drug-related stops are becoming more common over time.

The Boolean column drugs_related_stop indicates whether drugs were found during a given stop. I'll calculate the annual drug rate by resampling this column, and then I'll use a line plot to visualize how the rate has changed over time.

[39]:
```
# Save the annual rate of drug-related stops
annual_drug_rate = df.drugs_related_stop.resample('A').mean()

# Calculate the annual rate of drug-related stops
print(annual_drug_rate)
```

```
stop_datetime
2005-12-31    0.006501
2006-12-31    0.007258
2007-12-31    0.007970
2008-12-31    0.007505
2009-12-31    0.009889
2010-12-31    0.010081
2011-12-31    0.009731
2012-12-31    0.009921
2013-12-31    0.013094
2014-12-31    0.013826
2015-12-31    0.012266
Freq: A-DEC, Name: drugs_related_stop, dtype: float64
```

[40]:
```
# Create a line plot of 'annual_drug_rate'
annual_drug_rate.plot()

# Display the plot
plt.show()
```



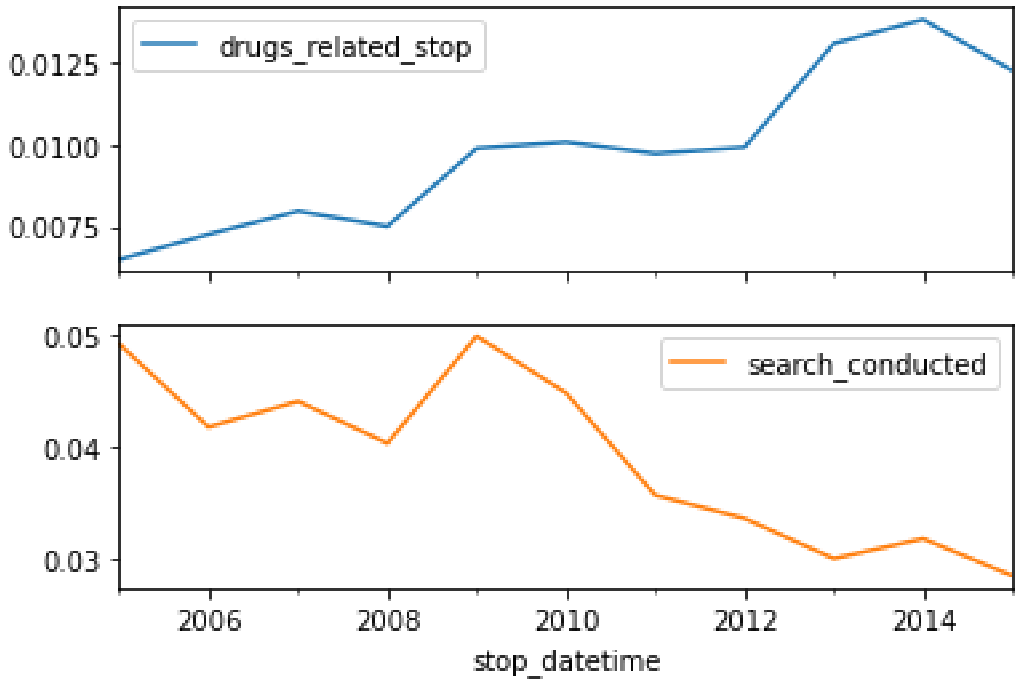
Interesting! The rate of drug-related stops nearly doubled over the course of 10 years. Why might that be the case?

3.2.1 Comparing drug and search rates

As per the finding above, the rate of drug-related stops increased significantly between 2005 and 2015. I might hypothesize that the rate of vehicle searches was also increasing, which would have led to an increase in drug-related stops even if more drivers were not carrying drugs.

I can test this hypothesis by calculating the annual search rate, and then plotting it against the annual drug rate. If the hypothesis is true, then I'll see both rates increasing over time.

[41]:
```
# Calculate and save the annual search rate
annual_search_rate = df.search_conducted.resample('A').mean()

# Concatenate 'annual_drug_rate' and 'annual_search_rate'
annual = pd.concat([annual_drug_rate, annual_search_rate], axis='columns')

# Create subplots from 'annual'
annual.plot(subplots = True)

# Display the subplots
plt.show()
```



The rate of drug-related stops increased even though the search rate decreased, disproving our hypothesis.

3.3 How long might you be stopped for a violation?

3.3.1 Converting stop durations to numbers

In the traffic stops dataset, the stop_duration column tells me approximately how long the driver was detained by the officer. Unfortunately, the durations are stored as strings, such as '0-15 Min'. How can I make this data easier to analyze?

In this exercise, I'll convert the stop durations to integers. Because the precise durations are not available, I'll have to estimate the numbers using reasonable values:

- Convert '0-15 Min' to 8
- Convert '16-30 Min' to 23
- Convert '30+ Min' to 45

[42]:
```
# Print the unique values in 'stop_duration'
print(df.stop_duration.unique())
```

```
['0-15 Min' '16-30 Min' '30+ Min']
```

[43]:
```
# Create a dictionary that maps strings to integers
mapping = {'0-15 Min': 8, '16-30 Min': 23, '30+ Min': 45}

# Convert the 'stop_duration' strings to integers using the 'mapping'
```

```
df['stop_minutes'] = df.stop_duration.map(mapping)

# Convert the 'stop_duration' strings to integers using the 'mapping'
df['stop_minutes'] = df.stop_duration.map(mapping)

# Print the unique values in 'stop_minutes'
print(df.stop_minutes.unique())
```

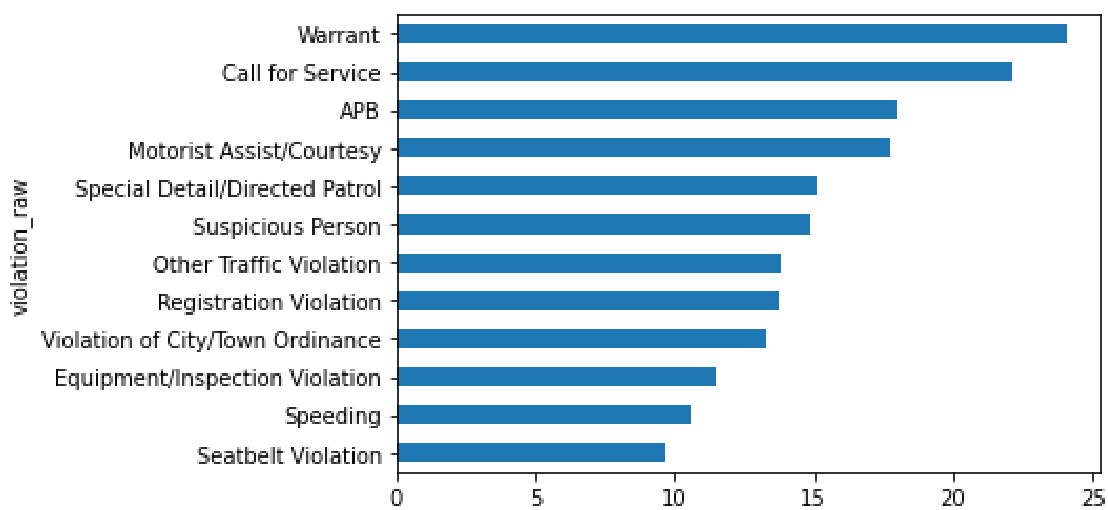
```
[ 8 23 45]
```

3.3.2 Plotting stop length

If I was stopped for a particular violation, how long might I expect to be detained?

I'll visualize the average length of time drivers are stopped for each type of violation. Rather than using the violation column in this exercise, I'll use violation_raw since it contains more detailed descriptions of the violations.

[44]:
```
# Calculate the mean 'stop_minutes' for each value in 'violation_raw'
stop_length = df.groupby('violation_raw')['stop_minutes'].mean()

# Sort 'stop_length' by its values and create a horizontal bar plot
stop_length.sort_values().plot(kind='barh')

# Display the plot
plt.show()
```



3.4 What violations are caught in each district?

3.4.1 Tallying violations by district

The state of Rhode Island is broken into six police districts, also known as zones. How do the zones compare in terms of what violations are caught by police?

I'll create a frequency table to determine how many violations of each type took place in each of the six zones. Then, I'll filter the table to focus on the "K" zone.

[45]:
```
# Create a frequency table of districts and violations
display(pd.crosstab(df.district, df.violation))

# Save the frequency table as 'all_zones'
all_zones = pd.crosstab(df.district, df.violation)

# Select rows 'Zone K1' through 'Zone K3'
display(all_zones.loc['Zone K1' :'Zone K3'])

# Save the smaller table as 'k_zones'
k_zones = all_zones.loc['Zone K1' :'Zone K3']
```

```
violation  Equipment  Moving violation  Other  Registration/plates  Seat belt  \
district
Zone K1          672              1254    290                  120          0
Zone K2         2061              2962    942                  768        481
Zone K3         2302              2898    705                  695        638
Zone X1          296               671    143                   38         74
Zone X3         2049              3086    769                  671        820
Zone X4         3541              5353   1560                 1411        843

violation  Speeding
district
Zone K1        5960
Zone K2       10448
Zone K3       12322
Zone X1        1119
Zone X3        8779
Zone X4        9795
```

```
violation  Equipment  Moving violation  Other  Registration/plates  Seat belt  \
district
Zone K1          672              1254    290                  120          0
Zone K2         2061              2962    942                  768        481
Zone K3         2302              2898    705                  695        638

violation  Speeding
district
Zone K1        5960
Zone K2       10448
```

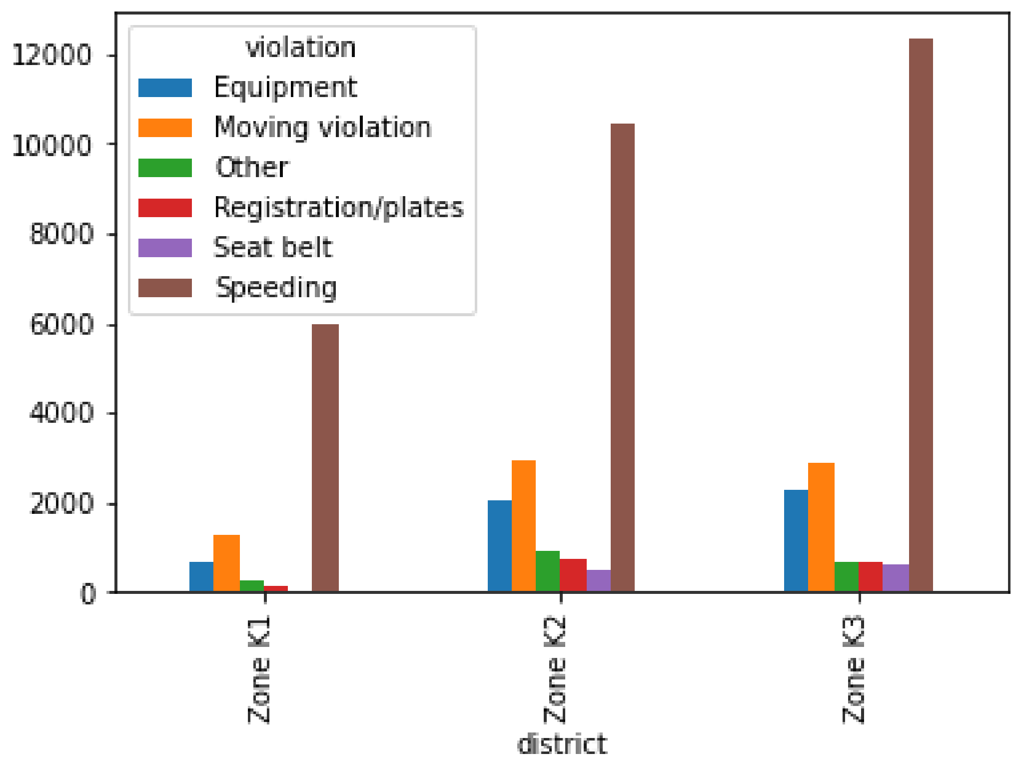
```
Zone K3         12322
```

3.4.2 Plotting violations by district

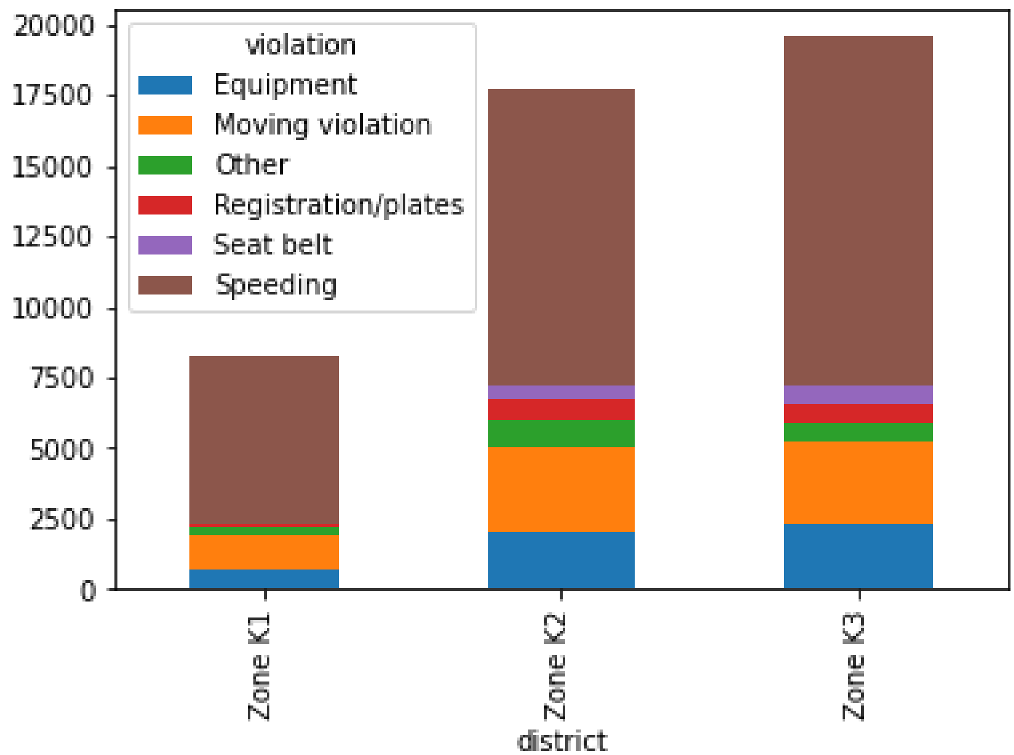
Now that I've created a frequency table focused on the "K" zones, I'll visualize the data to help me compare what violations are being caught in each zone.

First I'll create a bar plot, which is an appropriate plot type since I am comparing categorical data. Then I'll create a stacked bar plot in order to get a slightly different look at the data. Which plot do you find to be more insightful?

[46]:
```
# Create a bar plot of 'k_zones'
k_zones.plot(kind='bar')

# Display the plot
plt.show()
```



[47]:
```
# Create a stacked bar plot of 'k_zones'
k_zones.plot(kind='bar', stacked=True)

# Display the plot
plt.show()
```



Interesting! The vast majority of traffic stops in Zone K1 are for speeding, and Zones K2 and K3 are remarkably similar to one another in terms of violations.

4 Analyzing the effect of weather on policing

In this chapter, I will use a second dataset to explore the impact of weather conditions on police behavior during traffic stops. I will be merging and reshaping datasets, assessing whether a data source is trustworthy, working with categorical data, and other advanced skills.

4.1 Exploring the weather dataset

4.1.1 Plotting the temperature

[48]:
```
# Read 'weather.csv' into a DataFrame named 'weather'

weather_df=pd.read_csv('weather.csv')
weather_df.head()
```

[48]:
```
       STATION        DATE  AWND  TAVG  TMAX  TMIN  WT01  WT02  WT03  WT04  \
0  USC00379423  2005-01-01   NaN   NaN  47.0  28.0   NaN   NaN   NaN   NaN
1  USC00379423  2005-01-02   NaN   NaN  52.0  24.0   NaN   NaN   NaN   NaN
2  USC00379423  2005-01-03   NaN   NaN  48.0  27.0   NaN   NaN   NaN   NaN
3  USC00379423  2005-01-04   NaN   NaN  54.0  40.0   NaN   NaN   NaN   NaN
```

```
4  USC00379423  2005-01-05   NaN   NaN  44.0  31.0   NaN   NaN   NaN   NaN

   …  WT11  WT13  WT14  WT15  WT16  WT17  WT18  WT19  WT21  WT22
0  …   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN
1  …   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN
2  …   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN
3  …   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN
4  …   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN

[5 rows x 26 columns]
```

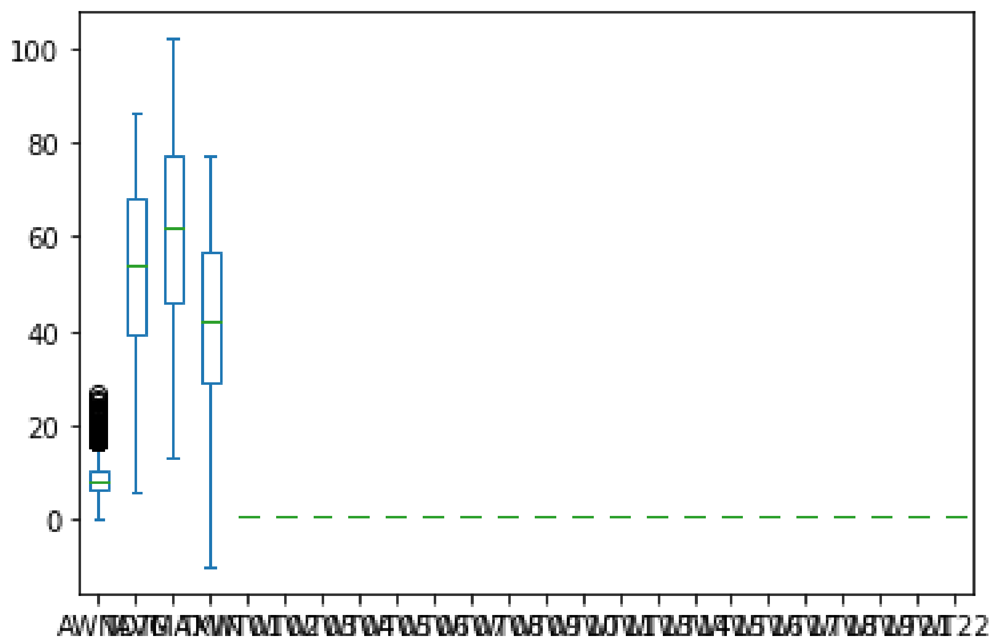
[49]:

```
# Read 'weather.csv' into a DataFrame named 'weather'

weather_df=pd.read_csv('weather.csv')
# Describe the temperature columns
print(weather_df[['TMIN', 'TAVG', 'TMAX']].describe())
```

```
              TMIN         TAVG         TMAX
count  7996.000000  1217.000000  8005.000000
mean     42.099425    52.493016    61.247096
std      17.386667    17.829792    18.495043
min     -10.000000     6.000000    13.000000
25%      29.000000    39.000000    46.000000
50%      42.000000    54.000000    62.000000
75%      57.000000    68.000000    77.000000
max      77.000000    86.000000   102.000000
```

[50]:

```
# Create a box plot of the temperature columns
weather_df.plot(kind='box')

# Display the plot
plt.show()
```



The temperature data looks good so far: the TAVG values are in between TMIN and TMAX, and the measurements and ranges seem reasonable.

4.1.2 Plotting the temperature difference

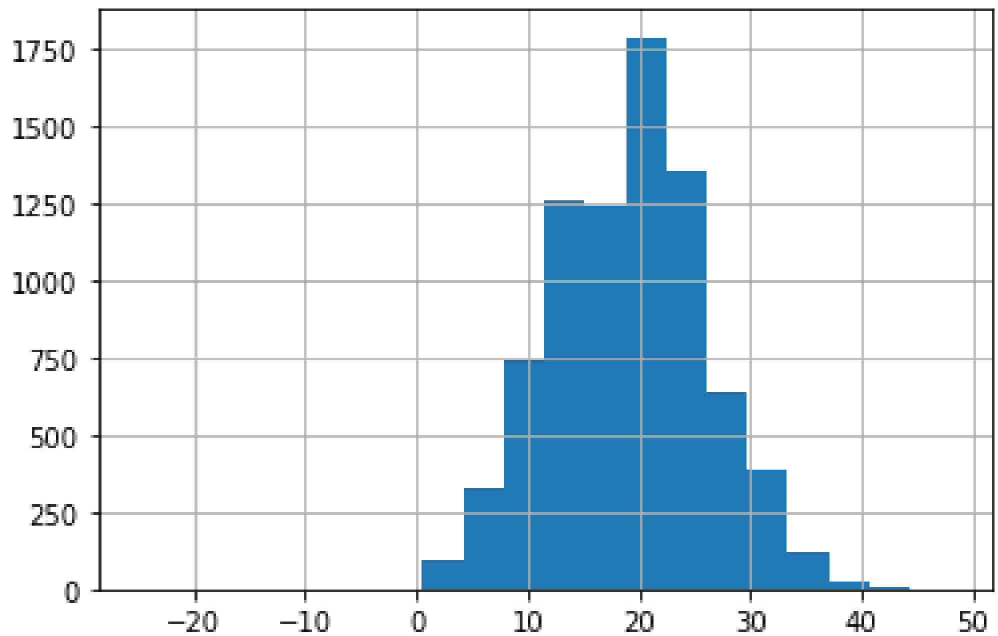
I'll continue to assess whether the dataset seems trustworthy by plotting the difference between the maximum and minimum temperatures.

What do I notice about the resulting histogram? Does it match my expectations, or do I see anything unusual?

[51]:
```
# Create a 'TDIFF' column that represents temperature difference
weather_df['TDIFF']=weather_df['TMAX']-weather_df['TMIN']

# Describe the 'TDIFF' column
print(weather_df.TDIFF.describe())
```

```
count    7994.000000
mean       19.149237
std         7.009716
min       -25.000000
25%        14.000000
50%        19.000000
75%        24.000000
max        48.000000
Name: TDIFF, dtype: float64
```

[52]:
```
# Create a histogram with 20 bins to visualize 'TDIFF'

weather_df.TDIFF.hist(bins=20)
# Display the plot
plt.show()
```



The TDIFF column has no negative values and its distribution is approximately normal, both of which are signs that the data is trustworthy.

4.2 Categorizing the weather

4.2.1 Counting bad weather conditions

The weather DataFrame contains 20 columns that start with 'WT', each of which represents a bad weather condition. For example:

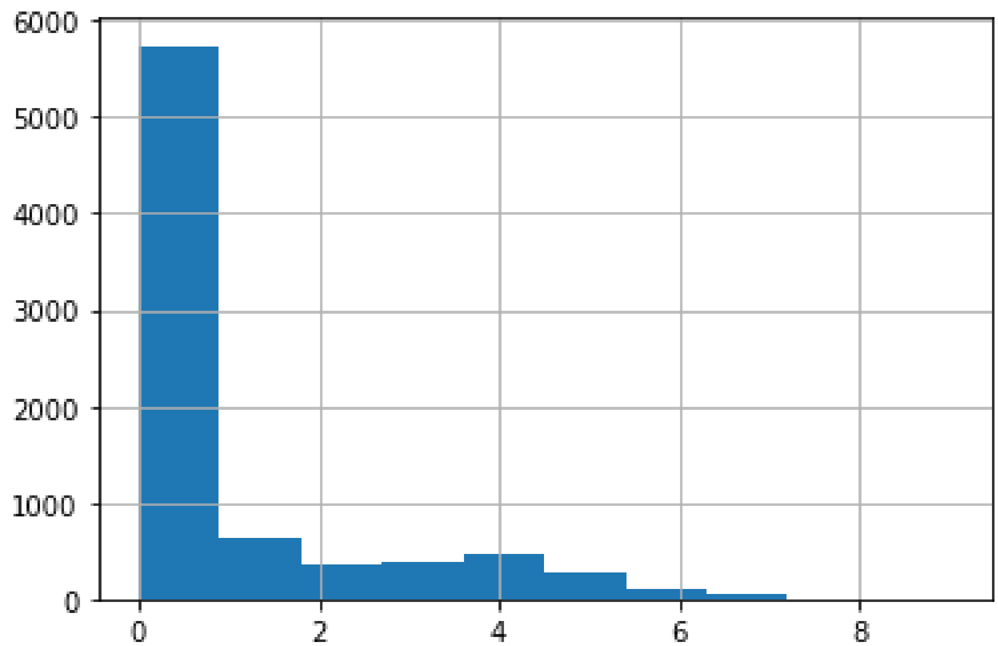
- WT05 indicates "Hail"
- WT11 indicates "High or damaging winds"
- WT17 indicates "Freezing rain"

For every row in the dataset, each WT column contains either a 1 (meaning the condition was present that day) or NaN (meaning the condition was not present).

Therefore, I'll quantify "how bad" the weather was each day by counting the number of 1 values in each row.

[53]:
```
# Copy 'WT01' through 'WT22' to a new DataFrame
WT = weather_df.loc[:, 'WT01':'WT22']
```

```
# Calculate the sum of each row in 'WT'
weather_df['bad_conditions'] = WT.sum(axis='columns')

# Replace missing values in 'bad_conditions' with '0'
weather_df['bad_conditions'] = weather_df.bad_conditions.fillna(0).astype('int')

# Create a histogram to visualize 'bad_conditions'
weather_df.bad_conditions.hist()

# Display the plot
plt.show()
```



It looks like many days didn't have any bad weather conditions, and only a small portion of days had more than four bad weather conditions.

4.2.2 Rating the weather conditions

previously, I counted the number of bad weather conditions each day. Now I'll use the counts to create a rating system for the weather.

The counts range from 0 to 9, and should be converted to ratings as follows:

- Convert 0 to 'good'
- Convert 1 through 4 to 'bad'
- Convert 5 through 9 to 'worse'

```
[54]: # Count the unique values in 'bad_conditions' and sort the index
print(weather_df.bad_conditions.value_counts().sort_index())

# Create a dictionary that maps integers to strings
mapping = {0:'good', 1:'bad', 2:'bad', 3:'bad', 4:'bad', 5:'worse',6:'worse',7:
 ↪'worse',8:'worse',9:'worse'}

# Convert the 'bad_conditions' integers to strings using the 'mapping'
weather_df['rating'] = weather_df.bad_conditions.map(mapping)

# Count the unique values in 'rating'
print(weather_df.rating.value_counts())
```

```
0    5738
1     628
2     368
3     380
4     476
5     282
6     101
7      41
8       4
9       4
Name: bad_conditions, dtype: int64
good     5738
bad      1852
worse     432
Name: rating, dtype: int64
```

4.2.3 Changing the data type to category

Since the rating column only has a few possible values, I'll change its data type to category in order to store the data more efficiently. I'll also specify a logical order for the categories, which will be useful for future analysis.

```
[55]: # Specify the logical order of the weather ratings
cats = pd.CategoricalDtype(['good', 'bad', 'worse'], ordered=True)

# Change the data type of 'rating' to category
weather_df['rating'] = weather_df.rating.astype(cats)

# Examine the head of 'rating'
print(weather_df.rating.head())
```

```
0    good
1    good
2    good
3    good
```

```
4    good
Name: rating, dtype: category
Categories (3, object): ['good' < 'bad' < 'worse']
```

4.3 Merging datasets

4.3.1 Preparing the DataFrames

I'll prepare the traffic stop and weather rating DataFrames so that they're ready to be merged:

With the df DataFrame, I'll move the stop_datetime index to a column since the index will be lost during the merge. With the weather DataFrame, I'll select the DATE and rating columns and put them in a new DataFrame.

[56]:
```python
# Reset the index of 'df'
df.reset_index(inplace=True)

# Examine the head of 'ri'
display(df.head())
```

```
         stop_datetime   stop_date stop_time driver_gender driver_race  \
0 2005-01-04 12:55:00  2005-01-04     12:55             M       White
1 2005-01-23 23:15:00  2005-01-23     23:15             M       White
2 2005-02-17 04:15:00  2005-02-17     04:15             M       White
3 2005-02-20 17:15:00  2005-02-20     17:15             M       White
4 2005-02-24 01:20:00  2005-02-24     01:20             F       White

                    violation_raw  violation  search_conducted search_type  \
0  Equipment/Inspection Violation  Equipment             False         NaN
1                        Speeding   Speeding             False         NaN
2                        Speeding   Speeding             False         NaN
3                Call for Service      Other             False         NaN
4                        Speeding   Speeding             False         NaN

    stop_outcome  is_arrested stop_duration  drugs_related_stop district  \
0       Citation        False      0-15 Min               False  Zone X4
1       Citation        False      0-15 Min               False  Zone K3
2       Citation        False      0-15 Min               False  Zone X4
3  Arrest Driver         True     16-30 Min               False  Zone X1
4       Citation        False      0-15 Min               False  Zone X3

   frisk  stop_minutes
0  False             8
1  False             8
2  False             8
3  False            23
4  False             8
```

[57]:
```
# Create a DataFrame from the 'DATE' and 'rating' columns
weather_rating=weather_df[['DATE', 'rating']]

# Examine the head of 'weather_rating'
display(weather_rating.head())
```

```
         DATE rating
0  2005-01-01   good
1  2005-01-02   good
2  2005-01-03   good
3  2005-01-04   good
4  2005-01-05   good
```

The df and weather_rating DataFrames are now ready to be merged

4.3.2 Merging the DataFrames

I'll merge the df and weather_rating DataFrames into a new DataFrame, df_weather.

The DataFrames will be joined using the stop_date column from df and the DATE column from weather_rating. Thankfully the date formatting matches exactly, which is not always the case!

Once the merge is complete, I'll set stop_datetime as the index, which is the column I saved in the previously.

[58]:
```
# Examine the shape of 'df'
print(df.shape)

# Merge 'df' and 'weather_rating' using a left join
df_weather = pd.merge(left=df, right=weather_rating, left_on='stop_date',␣
 ↪right_on='DATE', how='left')

# Examine the shape of 'df_weather'
display(df_weather.shape)

# Set 'stop_datetime' as the index of 'ri_weather'
df_weather.set_index('stop_datetime', inplace=True)
```

```
(86536, 16)
```

```
(172896, 18)
```

In the next section, I'll use df_weather to analyze the relationship between weather conditions and police behavior.

4.4 Does weather affect the arrest rate?

4.4.1 Comparing arrest rates by weather rating

Do police officers arrest drivers more often when the weather is bad? Let's find out!

- First, I'll calculate the overall arrest rate.

- Then, I'll calculate the arrest rate for each of the weather ratings you previously assigned.
- Finally, I'll add violation type as a second factor in the analysis, to see if that accounts for any differences in the arrest rate.

Since I previously defined a logical order for the weather categories, good < bad < worse, they will be sorted that way in the results.

[59]:
```
# Calculate the overall arrest rate
print(df_weather.is_arrested.mean())
```

```
0.03561678697020174
```

[60]:
```
# Calculate the arrest rate for each 'rating'
print(df_weather.groupby('rating').is_arrested.mean())
```

```
rating
good     0.035061
bad      0.036209
worse    0.041667
Name: is_arrested, dtype: float64
```

[61]:
```
# Calculate the arrest rate for each 'violation' and 'rating'
print(df_weather.groupby(['violation', 'rating']).is_arrested.mean())
```

```
violation            rating
Equipment            good      0.063118
                     bad       0.066212
                     worse     0.097357
Moving violation     good      0.057576
                     bad       0.058152
                     worse     0.065860
Other                good      0.078875
                     bad       0.086563
                     worse     0.062893
Registration/plates  good      0.088631
                     bad       0.098252
                     worse     0.115625
Seat belt            good      0.027302
                     bad       0.022243
                     worse     0.000000
Speeding             good      0.013647
                     bad       0.013202
                     worse     0.016886
Name: is_arrested, dtype: float64
```

The arrest rate increases as the weather gets worse, and that trend persists across many of the violation types. This doesn't prove a causal link, but it's quite an interesting result!

4.4.2 Selecting from a multi-indexed Series

The output of a single .groupby() operation on multiple columns is a Series with a MultiIndex. Working with this type of object is similar to working with a DataFrame:

- The outer index level is like the DataFrame rows.
- The inner index level is like the DataFrame columns.

I'll be accessing data from a multi-indexed Series using the .loc[] accessor.

[62]:
```python
# Save the output of the groupby operation from the last exercise
arrest_rate = df_weather.groupby(['violation', 'rating']).is_arrested.mean()

# Print the 'arrest_rate' Series
print(arrest_rate)

# Print the arrest rate for moving violations in bad weather
print(arrest_rate.loc['Moving violation', 'bad'])

# Print the arrest rates for speeding violations in all three weather conditions
print(arrest_rate.loc['Speeding'])
```

```
violation           rating
Equipment           good      0.063118
                    bad       0.066212
                    worse     0.097357
Moving violation    good      0.057576
                    bad       0.058152
                    worse     0.065860
Other               good      0.078875
                    bad       0.086563
                    worse     0.062893
Registration/plates good      0.088631
                    bad       0.098252
                    worse     0.115625
Seat belt           good      0.027302
                    bad       0.022243
                    worse     0.000000
Speeding            good      0.013647
                    bad       0.013202
                    worse     0.016886
Name: is_arrested, dtype: float64
0.058152027934461455
rating
good     0.013647
bad      0.013202
worse    0.016886
Name: is_arrested, dtype: float64
```

The .loc[] accessor is a powerful and flexible tool for data selection.

4.4.3 Reshaping the arrest rate data

I'll start by reshaping the arrest_rate Series into a DataFrame. This is a useful step when working with any multi-indexed Series, since it enables me to access the full range of DataFrame methods.

[64]:
```
# Unstack the 'arrest_rate' Series into a DataFrame
print(arrest_rate.unstack())
```

```
rating                   good       bad     worse
violation
Equipment            0.063118  0.066212  0.097357
Moving violation     0.057576  0.058152  0.065860
Other                0.078875  0.086563  0.062893
Registration/plates  0.088631  0.098252  0.115625
Seat belt            0.027302  0.022243  0.000000
Speeding             0.013647  0.013202  0.016886
```

5 Conclusion

I've now completed this analysis. Throughout the project, I used my pandas knowledge to prepare and analyze a dataset from start to finish. I cleaned messy data, created visualizations, answering questions about the data, and so much more.