**Exploratory Data Analysis Beginner Guide**

Data professionals use exploratory data analysis (EDA) to explore, study, and become familiar with a dataset's properties and the relationships between its variables. Data visualisation is one of the most important tools and approaches used in EDA. We can truly understand the data's appearance and the sorts of questions it may help us answer by analysing and visualising it using EDA. It also provides a means of identifying trends and patterns, identifying outliers and other abnormalities, and addressing certain important research problems.

These are some of the major steps carried out for this type of analysis:

1). Collect the data

2). Load the data

3). Get the basic information about the data

4). Handle duplicate values

5). Handle the unique values in the data

6). Visualize unique count

7). Find null values

8). Replace null values

9). Know the data type

10). Filter data

11). Get data's box plot

12). Get the basic information about the data

13). Create correlation plot

First, we make sure we are working with the correct libraries. For EDA, the following libraries are often used: pandas, numpy, seaborn and matplotlib. To do that, we use the functions:

Import pandas as pd

Import numpy as np

Import matplotlib.pyplot as plt

Import seaborn as sns

For the ease of convenience, we will use pyforest instead as shown below:

**import** pyforest

pd**.**set\_option( 'display.max\_columns', 200)

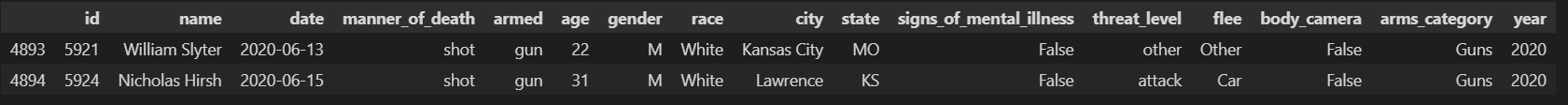
The dataset that we will be working on is the Police Shootings in USA. We can collect the data that will be used for analysis by using the code below:

df **=** pd**.**read\_csv("shootings.csv")

df.head (2) #To display the first 2 rows



df.tail(2) #To display the last 2 rows



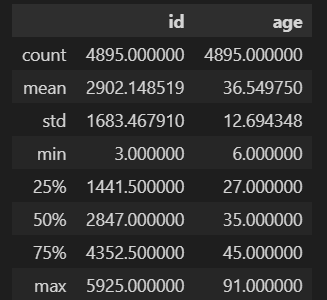
The next step would be to view what types of data we are dealing with at a glimpse.

df.shape #To show how many rows and columns are in the dataset

(4895, 16)

df.describe()

The describe function is very useful as it shows details such as mean, minimum and maximum values count among others



To get even more insight on our dataset, we can use df.info().

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 4895 entries, 0 to 4894

Data columns (total 15 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 id 4895 non-null int64

1 name 4895 non-null object

2 date 4895 non-null object

3 manner\_of\_death 4895 non-null object

4 armed 4895 non-null object

5 age 4895 non-null float64

6 gender 4895 non-null object

7 race 4895 non-null object

8 city 4895 non-null object

9 state 4895 non-null object

10 signs\_of\_mental\_illness 4895 non-null bool

11 threat\_level 4895 non-null object

12 flee 4895 non-null object

13 body\_camera 4895 non-null bool

14 arms\_category 4895 non-null object

dtypes: bool(2), float64(1), int64(1), object(11)

memory usage: 506.8+ KB

Here we are able to see vital info such as data types, entries, memory usage and any null values per column. We can also notice that ‘age’ column has a data type of float instead of int and ‘date’ column is represented as an object data type which is incorrect. This might cause potential issues in our analysis. However, there is an easy fix for this as shown below:

df['age'] = df['age'].astype(int)

df['date']=pd.to\_datetime(df['date'])

We run df.info() again and now all the data types are shown correctly.

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 4895 entries, 0 to 4894

Data columns (total 15 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 id 4895 non-null int64

1 name 4895 non-null object

2 date 4895 non-null datetime64[ns]

3 manner\_of\_death 4895 non-null object

4 armed 4895 non-null object

5 age 4895 non-null int32

6 gender 4895 non-null object

7 race 4895 non-null object

8 city 4895 non-null object

9 state 4895 non-null object

10 signs\_of\_mental\_illness 4895 non-null bool

11 threat\_level 4895 non-null object

12 flee 4895 non-null object

13 body\_camera 4895 non-null bool

14 arms\_category 4895 non-null object

dtypes: bool(2), datetime64[ns](1), int32(1), int64(1), object(10)

memory usage: 487.7+ KB

Next, we can confirm that there are no null values in our data.

df.isna().sum()

id 0

name 0

date 0

manner\_of\_death 0

armed 0

age 0

gender 0

race 0

city 0

state 0

signs\_of\_mental\_illness 0

threat\_level 0

flee 0

body\_camera 0

arms\_category 0

dtype: int64

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

0

It is also useful to view if we have any duplicated rows and remove them as they are redundant.

df.duplicated().sum()

0

Next, we will take some steps to visualize our data. The library matplotlib is useful in this step of our analysis. We would like to view the number of police shootings according to race in the USA.

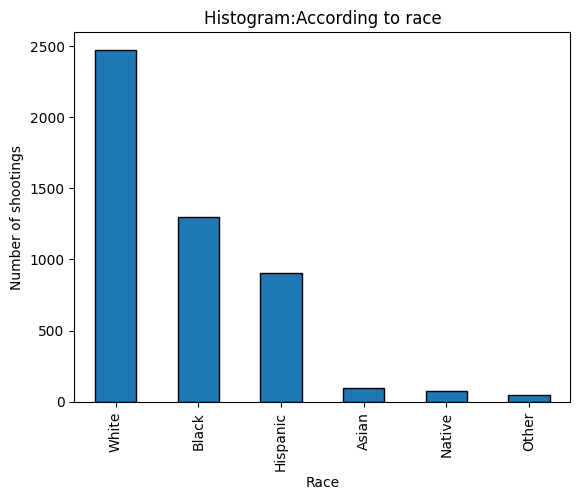
df['race'].value\_counts().plot(kind='bar', edgecolor = 'black')

plt.title('Histogram:According to race')

plt.xlabel('Race')

plt.ylabel('Number of shootings')

plt.show()



We can also visualize the number of police shootings according to age. For this we can use a histogram.

plt.hist(df['age'], bins=15, edgecolor = 'black')

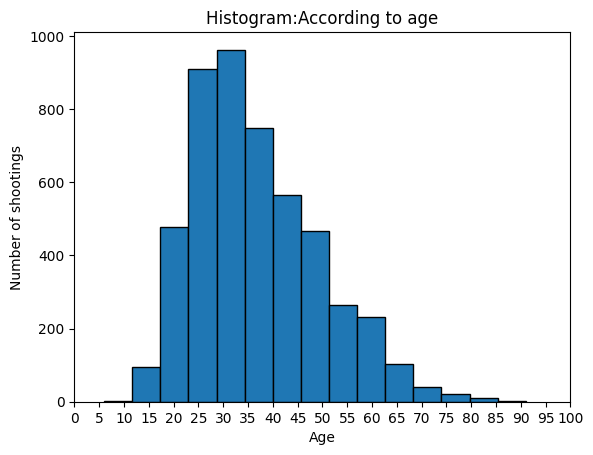
plt.title('Histogram:According to age')

plt.xlabel('Age')

plt.ylabel('Number of shootings')

plt.xticks( range ( 0, 101, 5))

plt.show()



The age group between 29-34 years seems to have encountered the most shootings by the police. To visualize police shootings according to year, we need to extract the year from the date column.

df['year'] = df['date'].dt.year

year\_shootings = df.groupby('year').size()

#count() can also be used instead of size()

Then using Pandas, we can use the groupby() function to group the data by year and calculate the number of shootings. The code snippet below can plot a line graph displaying the figures required.

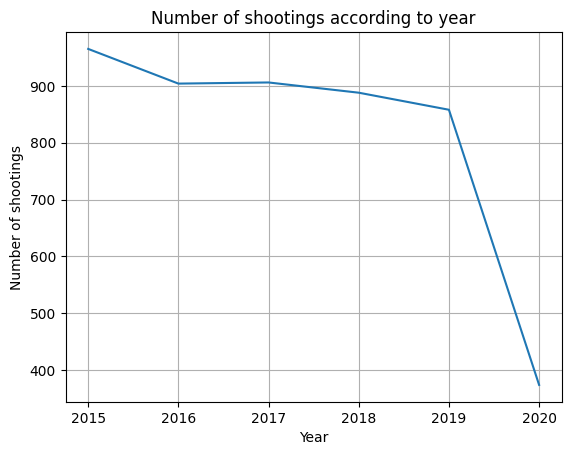
year\_shootings.plot (kind='line', grid = 'black')

plt.title('Number of shootings according to year')

plt.xlabel('Year')

plt.ylabel('Number of shootings')

plt.show()



We can observe that the number of police shootings had been steadily decreasing each year. In 2020, the figure drops significantly. Let’s do some inspection on the dataframe to identify why.

df.groupby('year').size()

year

2015 965

2016 904

2017 906

2018 888

2019 858

2020 374

dtype: int64

df.groupby(['year',df['date'].dt.month]).size()

year date

2015 1 75

2 77

3 91

4 83

5 69

..

2020 2 61

3 73

4 58

5 78

6 22

Length: 66, dtype: int64

df[df['year'] == 2019].groupby(['year', df['date'].dt.month]).size()

year date

2019 1 81

2 68

3 76

4 63

5 64

6 77

7 69

8 57

9 59

10 73

11 71

12 100

dtype: int64

df[df['year'] == 2020].groupby(['year', df['date'].dt.month]).size()

year date

2020 1 82

2 61

3 73

4 58

5 78

6 22

dtype: int64

It is now clear that our dataset only has the number of shootings upto June 2020 hence the drop for that year.

The seaborn library is built on top of the matplotlib library and can be used for powerful visualizations as well. Let’s visualize the number of police shootings according to age with a regression line as well.

First, we again use the groupby() function to group the data by age and then calculate the number of shootings by age.

df.groupby(df['age']) . size()

age

6 2

12 1

13 1

14 3

15 13

..

81 1

82 2

83 2

84 4

91 1

Length: 75, dtype: int64

age\_shootings = df.groupby(df['age']) . size()

sns.scatterplot(x=age\_shootings.index, y=age\_shootings.values)

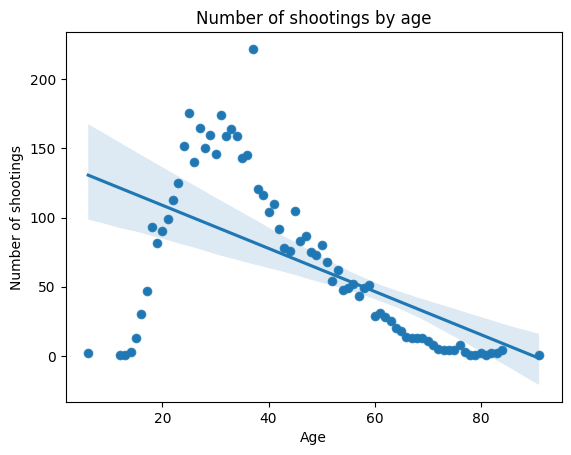
sns.regplot(x=age\_shootings.index, y=age\_shootings.values)

plt.title('Number of shootings by age')

plt.xlabel('Age')

plt.ylabel('Number of shootings')

The code creates a scatter plot of the number of shootings by age then regplot() function adds a regression line to the scatter plot. The regression line represents the best fit line that explains the relationship between the number of police shootings and age of the victims. We are able to better understand the trend of the data and the correlation between the 2 variables.



Next, we can visualize the number of police shootings according to race and by year.

shootings\_by\_race\_year = df.groupby(['year', 'race']).size()

shootings\_by\_race\_year = shootings\_by\_race\_year.unstack()

First, we use the groupby() function to group the data by year and race then calculate the total number of shootings as shown in the code above and use unstack() to present the data in a more readable format.

shootings\_by\_race\_year.plot(kind='bar', stacked=True)

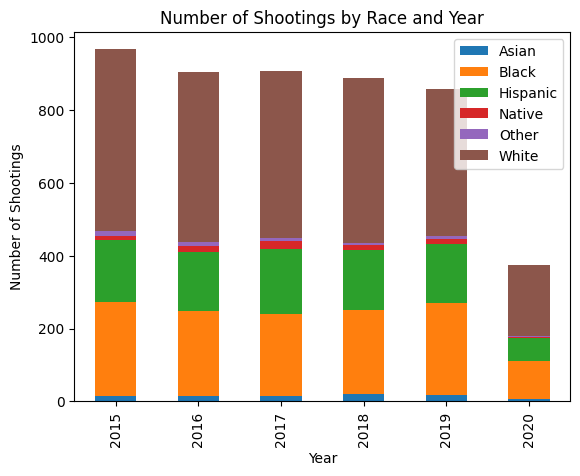
plt.xlabel('Year')

plt.ylabel('Number of Shootings')

plt.title('Number of Shootings by Race and Year')

plt.legend(loc='upper right')

plt.show()



As we can see, Exploratory Data Analysis is very crucial in the data science. The steps shown in this article are just some of the general steps taken in EDA.