report_code

August 12, 2020

1 Code

1.1 Traffic Cameras.ipynb (Traffic Cameras Analysis file)

1.1.1 Libraries and functions

[ ]:
```
%matplotlib inline
import matplotlib

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from csv import reader
import folium
```

1.1.2 Reading and merging data

[ ]:
```
#import traffic Camera location
camera_df = pd.read_csv('..\..\CSV_files\Traffic_Camera_Locations.csv')
camera_df['data']='TrafficCameraLocations'
#camera_df.head() #<-- visual QC of dataframe
```

[ ]:
```
#add traffic incidents
incidents_df = pd.read_csv('..\..\CSV_files\Traffic_Incidents.csv')
incidents_df['data']='Camera - Incident' #create a new column, to identify␣
↪which dataframe the data came from after the merge
#rename columns so they match other dataframes columns - makes it easier for␣
↪the merge
incidents_df=incidents_df.rename(columns={'Latitude':'latitude','Longitude':
↪'longitude','Count':'Incidents'})
#incidents_df.head() #<-- visual QC of dataframe
```

[ ]:
```
#data given to 6 significant digets. Here we reduce it to 4, so that camera␣
↪data will be matched up with accidents that occured
#within ~10m's
incidents_df = incidents_df.round({'latitude':4, 'longitude':4})
camera_df = camera_df.round({'latitude':4, 'longitude':4})
#camera_df.head() #<-- visual QC of dataframe
```

[ ]:
```
#merging two dataframes into df_total. merging on camera_df so we can determine␣
↪which cameras has an incident occur near them
df_total = pd.merge(left=incidents_df, right=camera_df, how='right',␣
↪left_on=['latitude','longitude'], right_on=['latitude','longitude'])
#nan data in "data_x" column belong to camera rows that had no incidents
#so replace those nan values with "Camera - no incident", so we can "group-by"␣
↪later to count number of "camera no incident"
# and "camera - incident".
df_total['data_x'] = df_total['data_x'].fillna("Camera - no incident")
#need to fill "incidents" with a "1" value for even the non-incident rows, so␣
↪that we have something to count when we 'groupby'
df_total['Incidents'] = df_total['Incidents'].fillna(1)
df_total= df_total.rename(columns={"Incidents":"Count"})  #rename incidents to␣
↪count, so the title makes more sense
#df_total.head() #<-- visual QC of dataframe
```

1.1.3 Table showing the percentage of cameras that caught an incident (within ~10m of a camera (Table))

[ ]:
```
#'groupby-sum' /(count) the number of incident rows, and non-incident rows. and␣
↪put into a nice table
df_total = df_total.groupby(['data_x']).sum()
#df_total   #4 decimal points =~10m radius around camera latitudes and␣
↪longitudes
```

[ ]:
```
#It would be nice to calculate % of cameras that caught incidents in the above␣
↪table
#To do that, need to calculate total first
totalCount = df_total['Count'].sum()
#totalCount #<-- visual QC check
df_total['%'] = (df_total['Count']/totalCount)*100
df_total = df_total[['Count','%']]
df_total #<-- show table
```

1.1.4 Cameras near incidents (bar graph)

[ ]:
```
#create histogram plot of above data
df_total=df_total.reset_index()
fig,ax = plt.subplots(figsize=(15,7))

sns.barplot(x = 'data_x', y = '%', data = df_total)

plt.ylabel("Percentage (%)",fontsize=20)
plt.xlabel("Cameras",fontsize=20)
plt.title("Cameras that Caught Incidents",fontsize=24)
```

1.1.5 Traffic Cameras (part 2)

[ ]:
```
# Re-Doing above tables and graphs, but merging onto "incident dataframe", so␣
↪that we can determine
# how many incidents occured near cameras (oposite of the above)
incidents_df = incidents_df.round({'latitude':4, 'longitude':4}) #rounding lat␣
↪and long aloud them to be grouped wihtin a 10m accuracy
camera_df = camera_df.round({'latitude':4, 'longitude':4})      #this is nice,␣
↪because accidents and cameras dont need to occur exactly ontop of eachother
camera_df.data = 'Incidents with Cameras' #change input calues of 'data' column␣
↪to set up for histogram plot later
#camera_df#<-- visual QC of dataframe
```

[ ]:
```
#merge data on traffic incidents this time. so we have the total amount of␣
↪incidents, but only some of them have camera data
#this allows us to compare how many incidents had cameras near them
df_total = pd.merge(left=incidents_df, right=camera_df, how='left',␣
↪left_on=['latitude','longitude'], right_on=['latitude','longitude'])
df_total['data_x'] = df_total['data_x'].fillna("Camera")
df_total['data_y'] = df_total['data_y'].fillna('Incidents with NO Cameras')␣
↪#filled na's with a proper name, so we can groupby.
df_total= df_total.rename(columns={"Incidents":"Count"})
#df_total.tail(20)#<-- visual QC of dataframe
```

[ ]:
```
totalCount = df_total['Count'].sum() #calculate the total rows, so we can␣
↪calculate percentage
#totalCount #<-- visual QC of output
```

1.1.6 Number and percentage of incidents that occur with and without a camera nearby (Table)

[ ]:
```
df_total2 = df_total.groupby(['data_y']).sum()
df_total2['%'] = (df_total2['Count']/totalCount)*100
df_total2 = df_total2[['Count','%']]
#JOINED ONTO INCIDENTS, so we see only incident rows, and can see how many have␣
↪cameras
df_total2 #<-- show table
```

1.1.7 Incidents that occured with and without a camera nearby (bar graph)

[ ]:
```
#create histogram
df_total2=df_total2.reset_index()
fig,ax = plt.subplots(figsize=(15,7))
sns.barplot(x = 'data_y', y = '%', data = df_total2)

plt.ylabel("Percentage (%)",fontsize=20)
plt.xlabel("Incidents",fontsize=20)
plt.title("Incidents Caught by Camera",fontsize=24)
```

1.2 Traffic Signals.ipynb (Traffic Signals Analysis)

1.2.1 Libraries and functions

[ ]:
```
import numpy as np
import pandas as pd
import folium
import geopandas as gpd
from shapely.geometry import Polygon
from shapely.geometry import box
import matplotlib.pyplot as plt
import seaborn as sns
```

1.2.2 Reading and merging data

[ ]:
```
#add traffic incidents
incidents_df = pd.read_csv('..\..\CSV_files\Traffic_Incidents.csv') #call csv
incidents_df['data']='TrafficIncidents' #create column to keep track of where␣
↪this data came from when merging dataframes
#rename columns so they match other dataframes
incidents_df=incidents_df.rename(columns={'Latitude':'latitude','Longitude':
↪'longitude','Count':'Incidents'})
#incidents_df.head() #<-- visual QC of dataframe
```

[ ]:
```
#data given to 6 significant digets, so this doesnt actualy do much(but we kept␣
↪the code, just incase we wanted to
#experiment with radius)
incidents_df = incidents_df.round({'latitude':6, 'longitude':6})
#incidents_df #<-- visual QC of dataframe
```

[ ]:
```
#add trafic signals
signals_df = pd.read_csv('..\..\CSV_files\Traffic_Signals.csv')
signals_df=signals_df.rename(columns={'Count':'Signal Count'})
signals_df['data']='TrafficSignals'
#signals_df #<-- visual QC of dataframe
```

[ ]:
```
signals_df = signals_df.round({'latitude':6, 'longitude':6})
```

[ ]:
```
#DID A LEFT JOIN, BECASUE MORE SIGNALS THAN ACCIDENTS DATA
accidents_vs_signals = pd.merge(left=signals_df, right=incidents_df,␣
↪how='left', left_on=['latitude','longitude'],␣
↪right_on=['latitude','longitude'])
#accidents_vs_signals #<-- visual QC of dataframe
```

1.2.3 Incidents and signal count vs. intersection type (Table)

[ ]:
```
INT_TYPE_TABLE = accidents_vs_signals.groupby(['INT_TYPE']).sum()
INT_TYPE_TABLE = INT_TYPE_TABLE[INT_TYPE_TABLE.columns.
 ↪difference(['latitude','longitude','ACCESSIBLE PEDESTRIAN SIGNAL'])]
INT_TYPE_TABLE['%Incidents to Signal Count'] = (INT_TYPE_TABLE['Incidents'] /␣
 ↪INT_TYPE_TABLE['Signal Count'])*100
INT_TYPE_TABLE = INT_TYPE_TABLE.sort_values(['%Incidents to Signal Count'],␣
 ↪ascending=False) #order table
INT_TYPE_TABLE #<---print table for analysis
```

1.2.4 Percentatge of incidents vs. signal type (Histogram)

[ ]:
```
#create histogram of the above table
INT_TYPE_TABLE=INT_TYPE_TABLE.reset_index()
fig,ax = plt.subplots(figsize=(15,7))
sns.barplot(x = 'INT_TYPE', y = '%Incidents to Signal Count', data =␣
 ↪INT_TYPE_TABLE)

plt.ylabel("% Incidents to Signal Count",fontsize=20)
plt.xlabel("Signal Type",fontsize=20)
plt.title("Percentatge of Incidents vs. Signal Type",fontsize=24)
```

1.2.5 Incident to signal count percentage in each city quadrant (Table)

[ ]:
```
QUADRANT_TABLE = accidents_vs_signals.groupby(['QUADRANT_x']).sum()
QUADRANT_TABLE = QUADRANT_TABLE[QUADRANT_TABLE.columns.
 ↪difference(['latitude','longitude','ACCESSIBLE PEDESTRIAN SIGNAL'])]
QUADRANT_TABLE['%Incidents to Signal Count'] = (QUADRANT_TABLE['Incidents'] /␣
 ↪QUADRANT_TABLE['Signal Count'])*100
QUADRANT_TABLE = QUADRANT_TABLE.sort_values(['%Incidents to Signal Count'],␣
 ↪ascending=False)
QUADRANT_TABLE=QUADRANT_TABLE.reset_index()
QUADRANT_TABLE #<-- show to see table
```

[ ]:
```
#create histogram of the above table
fig,ax = plt.subplots(figsize=(15,7))
sns.barplot(x = 'QUADRANT_x', y = '%Incidents to Signal Count', data =␣
 ↪QUADRANT_TABLE)

plt.ylabel("% Incidents to Signal Count",fontsize=20)
plt.xlabel("City Quadrant",fontsize=20)
plt.title("Incident to Signal Count Percentage in each City␣
 ↪Quadrant",fontsize=24)
```

1.2.6 Pedestrian button at intersection correlation (Table)

[ ]:
```
PEDBUTTON_TABLE = accidents_vs_signals.groupby(['PEDBUTTONS']).sum()
PEDBUTTON_TABLE = PEDBUTTON_TABLE[PEDBUTTON_TABLE.columns.
 ↪difference(['latitude','longitude','ACCESSIBLE PEDESTRIAN SIGNAL'])]
PEDBUTTON_TABLE['%Incidents to Signal Count'] = (PEDBUTTON_TABLE['Incidents'] /␣
 ↪PEDBUTTON_TABLE['Signal Count'])*100
PEDBUTTON_TABLE=PEDBUTTON_TABLE.reset_index()
PEDBUTTON_TABLE #<-- show to see table
```

1.2.7 Incident per signal count percentage for each quadrant in Calgary (BubblePlot)

[ ]:
```
x=[0,1,0,0.5,1]
y=[0,1,1,0,0]
z=[1670,1020,394,4130,1357]
plt.scatter(x,y,s=z, c="red", alpha=0.5)

plt.ylabel("South           North",fontsize=20)
plt.xlabel("West                               East",fontsize=20)
plt.title("Incident per Signal Count Percentage for each Quadrant in␣
 ↪Calgary",fontsize=24)
```

1.2.8 Incidents vs. signal count for each quadrant in Calgary (pointplot)

[ ]:
```
#bins = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24]

fig,ax = plt.subplots(figsize=(15,7))
sns.set(style="darkgrid")
g = sns.pointplot(x="Signal Count", y="Incidents", data=QUADRANT_TABLE,␣
 ↪color="black")
g = sns.pointplot(x="Signal Count", y="Incidents", hue='QUADRANT_x',␣
 ↪data=QUADRANT_TABLE, scale=2)

plt.ylabel("Incidents" ,fontsize=20)
plt.xlabel("Signal Count",fontsize=20)
plt.title("Incidents vs. Signal Count for each Quadrant in Calgary",fontsize=24)
```

1.3 Weather Analysis.ipynb (Weather Analysis File)

[ ]:
```
import numpy as np
import pandas as pd
import folium
import geopandas as gpd
from shapely.geometry import Polygon
from shapely.geometry import box
import matplotlib.pyplot as plt
import seaborn as sns
```

1.3.1 Reading and merging data

[ ]:
```
#add traffic incidents
incidents_df = pd.read_csv('..\..\CSV_files\Traffic_Incidents.csv') #get data␣
 ↪from CSV file
incidents_df['data']='TrafficIncidents' #create "data" column to record which␣
 ↪dataframe this data came from for later
#rename columns so they match other dataframes
incidents_df=incidents_df.rename(columns={'Latitude':'latitude','Longitude':
 ↪'longitude'})
#incidents_df.head() #<-- used for visual QC
```

[ ]:
```
#split Start_DT into Day/Month/Year (because only interested in 2018)
incidents_df = pd.DataFrame(incidents_df)
incidents_df['MonthDayYear']=incidents_df['START_DT'].str[:10] #re-arrage␣
 ↪columns values, and rename
#incidents_df.head() #<-- used for visual QC
```

[ ]:
```
incidents_df = incidents_df.
 ↪groupby('MonthDayYear')['Count','longitude','latitude','MonthDayYear'].sum()
incidents_df['MonthDayYear']= incidents_df.index
#incidents_df #<-- for visual QC (dont worry, there will be a python warning␣
 ↪when this code executes, that is because
#                    the index has the same title as another column in the␣
 ↪dataframe, but this column will soon
#                     me altered in later steps)
```

[ ]:
```
#Seperating the Year,Day and Month from a combined column in the DataFrame.␣
 ↪This later allows us to plot more interesting plots
incidents_df = pd.DataFrame(incidents_df)
incidents_df['Year']=incidents_df['MonthDayYear'].str[6:10]
incidents_df = incidents_df.loc[incidents_df['Year'] == '2018'] #only␣
 ↪interested in 2018
incidents_df['Day']=incidents_df['MonthDayYear'].str[3:5]
incidents_df['Month']=incidents_df['MonthDayYear'].str[:2]
```

```
#incidents_df #<-- used to visualy QC DataFrame
```

[ ]:
```
#need to change datatypes of Year, Month and Day from Object to Integer for the␣
↪merge of dataframes
incidents_df['Year']= incidents_df['Year'].astype(str).astype(int)
incidents_df['Month']= incidents_df['Month'].astype(str).astype(int)
incidents_df['Day']= incidents_df['Day'].astype(str).astype(int)
```

[ ]:
```
#Lets get the Weather Daily Data Now
pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', None)

# returns a DataFrame with weather data from "climate.weather.gc.ca"
def download_weather_data(station, year, month=1, daily=True):

    # url to retrieve hourly data
    url_template_hourly = "https://climate.weather.gc.ca/climate_data/
↪bulk_data_e.html?
↪format=csv&stationID={station}&Year={year}&Month={month}&Day=14&timeframe=1&submit=Download

    # url to retrieve daily data
    url_template_daily = "https://climate.weather.gc.ca/climate_data/
↪bulk_data_e.html?
↪format=csv&stationID={station}&Year={year}&Month={month}&Day=14&timeframe=2&submit=Download

    # daily data by default
    if(daily == True):
        url = url_template_daily.format(station=station, year=year, month=month)

    # hourly data when (daily == False)
    else:
        url = url_template_hourly.format(station=station, year=year,␣
↪month=month)

    # read data into dataframe, use headers and set Date/Time column as index
    weather_data = pd.read_csv(url, index_col='Date/Time', parse_dates=True)

    # replace the degree symbol in the column names
    weather_data.columns = [col.replace('\xb0', '') for col in weather_data.
↪columns]

    return weather_data
```

[ ]:
```
df = download_weather_data(50430, 2018) #<-- get hourly weather
```

[ ]:
```
#rename some columns to make them match other dataframes
df = df.rename(columns={"Longitude (x)":"longitude" , "Latitude (y)":
↪"latitude"})
Daily_Weather = df[['latitude','longitude','Station␣
↪Name','Year','Month','Day','Mean Temp (C)']]
Daily_Weather['data']='Weather'
#Daily_Weather <-- visual QC
```

[ ]:
```
Hourly_Weather = download_weather_data(50430, 2018, daily=False) # <-- get␣
↪hourly Data (visibility)
#rename some columns to make them match other dataframes, also add column to␣
↪record where this data came from for after merge
Hourly_Weather = Hourly_Weather.rename(columns={"Longitude (x)":"longitude" ,␣
↪"Latitude (y)":"latitude"})
Hourly_Weather['data']='Weather'
#Hourly_Weather.head() #<-- visual QC
```

[ ]:
```
#groupby.mean() on "Day" column, to get the average visibulity for each day␣
↪(daily data was requested for thsi report)
Hourly_Weather = Hourly_Weather.
↪groupby('Day')['longitude','latitude','Visibility␣
↪(km)','Year','Month','Day'].mean()
#Hourly_Weather.head() #<-- will get a warning because index has same name as a␣
↪column, but it is okay for now.
```

[ ]:
```
#continue with above strategy, pulling in all weather data for each month
Hourly_Weather2 = download_weather_data(50430, 2018, month = 2, daily=False)
Hourly_Weather2 = Hourly_Weather2.rename(columns={"Longitude (x)":"longitude" ,␣
↪"Latitude (y)":"latitude"})
Hourly_Weather2['data']='Weather'
Hourly_Weather2 = Hourly_Weather2.
↪groupby('Day')['longitude','latitude','Visibility␣
↪(km)','Year','Month','Day'].mean()

Hourly_Weather3 = download_weather_data(50430, 2018, month = 3, daily=False)
Hourly_Weather3 = Hourly_Weather3.rename(columns={"Longitude (x)":"longitude" ,␣
↪"Latitude (y)":"latitude"})
Hourly_Weather3['data']='Weather'
Hourly_Weather3 = Hourly_Weather3.
↪groupby('Day')['longitude','latitude','Visibility␣
↪(km)','Year','Month','Day'].mean()

Hourly_Weather4 = download_weather_data(50430, 2018, month = 4, daily=False)
Hourly_Weather4 = Hourly_Weather4.rename(columns={"Longitude (x)":"longitude" ,␣
↪"Latitude (y)":"latitude"})
Hourly_Weather4['data']='Weather'
```

```
Hourly_Weather4 = Hourly_Weather4.
↪groupby('Day')['longitude','latitude','Visibility␣
↪(km)','Year','Month','Day'].mean()

Hourly_Weather5 = download_weather_data(50430, 2018, month = 5, daily=False)
Hourly_Weather5 = Hourly_Weather5.rename(columns={"Longitude (x)":"longitude" ,␣
↪"Latitude (y)":"latitude"})
Hourly_Weather5['data']='Weather'
Hourly_Weather5 = Hourly_Weather5.
↪groupby('Day')['longitude','latitude','Visibility␣
↪(km)','Year','Month','Day'].mean()

Hourly_Weather6 = download_weather_data(50430, 2018, month = 6, daily=False)
Hourly_Weather6 = Hourly_Weather6.rename(columns={"Longitude (x)":"longitude" ,␣
↪"Latitude (y)":"latitude"})
Hourly_Weather6['data']='Weather'
Hourly_Weather6 = Hourly_Weather6.
↪groupby('Day')['longitude','latitude','Visibility␣
↪(km)','Year','Month','Day'].mean()

Hourly_Weather7 = download_weather_data(50430, 2018, month = 7, daily=False)
Hourly_Weather7 = Hourly_Weather7.rename(columns={"Longitude (x)":"longitude" ,␣
↪"Latitude (y)":"latitude"})
Hourly_Weather7['data']='Weather'
Hourly_Weather7 = Hourly_Weather7.
↪groupby('Day')['longitude','latitude','Visibility␣
↪(km)','Year','Month','Day'].mean()

Hourly_Weather8 = download_weather_data(50430, 2018, month = 8, daily=False)
Hourly_Weather8 = Hourly_Weather8.rename(columns={"Longitude (x)":"longitude" ,␣
↪"Latitude (y)":"latitude"})
Hourly_Weather8['data']='Weather'
Hourly_Weather8 = Hourly_Weather8.
↪groupby('Day')['longitude','latitude','Visibility␣
↪(km)','Year','Month','Day'].mean()

Hourly_Weather9 = download_weather_data(50430, 2018, month = 9, daily=False)
Hourly_Weather9 = Hourly_Weather9.rename(columns={"Longitude (x)":"longitude" ,␣
↪"Latitude (y)":"latitude"})
Hourly_Weather9['data']='Weather'
Hourly_Weather9 = Hourly_Weather9.
↪groupby('Day')['longitude','latitude','Visibility␣
↪(km)','Year','Month','Day'].mean()

Hourly_Weather10 = download_weather_data(50430, 2018, month = 10, daily=False)
```

```
Hourly_Weather10= Hourly_Weather10.rename(columns={"Longitude (x)":"longitude"␣
↪, "Latitude (y)":"latitude"})
Hourly_Weather10['data']='Weather'
Hourly_Weather10 = Hourly_Weather10.
↪groupby('Day')['longitude','latitude','Visibility␣
↪(km)','Year','Month','Day'].mean()

Hourly_Weather11 = download_weather_data(50430, 2018, month = 11, daily=False)
Hourly_Weather11 = Hourly_Weather11.rename(columns={"Longitude (x)":"longitude"␣
↪, "Latitude (y)":"latitude"})
Hourly_Weather11['data']='Weather'
Hourly_Weather11 = Hourly_Weather11.
↪groupby('Day')['longitude','latitude','Visibility␣
↪(km)','Year','Month','Day'].mean()

Hourly_Weather12 = download_weather_data(50430, 2018, month = 12, daily=False)
Hourly_Weather12 = Hourly_Weather12.rename(columns={"Longitude (x)":"longitude"␣
↪, "Latitude (y)":"latitude"})
Hourly_Weather12['data']='Weather'
Hourly_Weather12 = Hourly_Weather12.
↪groupby('Day')['longitude','latitude','Visibility␣
↪(km)','Year','Month','Day'].mean()

#combined all data
hourly_Weather_Total = pd.
↪concat([Hourly_Weather,Hourly_Weather2,Hourly_Weather3,Hourly_Weather4,Hourly_Weather5,\
                                  Hourly_Weather6,Hourly_Weather7,␣
↪Hourly_Weather8, Hourly_Weather9, Hourly_Weather10,\
                                 Hourly_Weather11, Hourly_Weather12])
#hourly_Weather_Total
```

[ ]:
```
#JOIN HOURLY WEATHER TO DAILY WEATHER
Daily_Weather.index.name=None
hourly_Weather_Total.index.name=None
total_weather = pd.merge(left=hourly_Weather_Total, right=Daily_Weather,␣
↪how='outer', left_on=['Year','Month','Day'], right_on=['Year','Month','Day'])
#total_weather.head() #<-- QC check
```

1.3.2 Table of general statistics of temperature and visibility (Table)

```
[ ]: analysis = total_weather[['Visibility (km)','Mean Temp (C)']] # only keep␣
      ↪columns we are interested in
     analysis.describe() # <-- unhide to see table of temperature and visibility␣
      ↪statistics in 2018
```

```
[ ]: #combine Total Weather with collition Data
     Colition_weather_df = pd.merge(left=total_weather, right=incidents_df,␣
      ↪how='left', left_on=['Year','Month','Day'], right_on=['Year','Month','Day'])
     Colition_weather_df=Colition_weather_df.reset_index()
     #Colition_weather_df #<-- visual QC
     #Colition_weather_df.to_csv("C:/Users/adamd/Desktop/WeatherIncidents.csv") <-␣
      ↪hard copy QC
```

1.3.3 Number of days at each temperature (Histogram)

```
[ ]: bins = [-30,-25,-20,-15,-10,-5,0,5,10,15,20,25,30]

     fig,ax = plt.subplots(figsize=(15,7))
     sns.distplot(Daily_Weather['Mean Temp (C)'],kde=False , bins = bins,␣
      ↪hist_kws={"rwidth":0.8,'edgecolor':'black', 'alpha':1.0} )
     plt.ylabel("Days at Temperature",fontsize=20)
     plt.xlabel("Temperature (C)",fontsize=20)
     plt.title("Number of days at each Temperature",fontsize=24)
```

1.3.4 Daily incidents vs. binned mean temperature (C) (PointPlot)

```
[ ]: # Your solution goes here
     bins = [-30,-25,-20,-15,-10,-5,0,5,10,15,20,25,30]
     Colition_weather_df['Temp Binned'] = pd.cut(Colition_weather_df['Mean Temp␣
      ↪(C)'], bins=bins)

     fig,ax = plt.subplots(figsize=(15,7))
     g = sns.pointplot(x="Temp Binned", y="Count", data=Colition_weather_df , ci =␣
      ↪None)
     g.set(ylim=(0, 50))

     plt.ylabel("Daily Incidents",fontsize=20)
     plt.xlabel("Binned Mean Temperature (C)",fontsize=20)
     plt.title("Daily Incidents vs. Binned Mean Temperature (C)",fontsize=24)
     #Even though it looks like more colisions might happen around the temperatures␣
      ↪of -10 and +10, it is only because there are
     #more days at those temperatures (see above). However % wise, there are very␣
      ↪few days netween -20 and -20 C, yes we still have
     #on average 13 to 16 incidents
```

1.3.5 Mean temperature (C) per day (PointPlot)

```
[ ]: fig,ax = plt.subplots(figsize=(15,7))
     sns.set(style="darkgrid")
     g = sns.pointplot(x="index", y="Mean Temp (C)", data=Colition_weather_df,␣
      ↪ci=None)
     ax.xaxis.set_major_formatter(plt.NullFormatter())

     plt.ylabel("Mean Temperature (C)",fontsize=20)
     plt.xlabel("Jan          Feb          Mar          Apr          May          Jun          ␣
      ↪Jul          Aug          Sep          Oct          Nov          Dec",fontsize=20)
     plt.title("Mean Temperature (C) per Day",fontsize=24)
```

1.3.6 Mean temperature (C) per month (PointPlot)

```
[ ]: #PLotting some daily Weather Conditions (Temp and Visibility)
     fig,ax = plt.subplots(figsize=(15,7))
     sns.set(style="darkgrid")
     g = sns.pointplot(x="Month", y="Mean Temp (C)", data=Colition_weather_df,␣
      ↪ci=None)

     plt.ylabel("Mean Temperature (C)",fontsize=20)
     plt.xlabel("Month",fontsize=20)
     plt.title("Mean Temperature (C) per Month",fontsize=24)
```

1.3.7 Daily incidents per month (PointPlot)

```
[ ]: #PLotting some daily Weather Conditions (Temp and Visibility)
     fig,ax = plt.subplots(figsize=(15,7))
     sns.set(style="darkgrid")
     g = sns.pointplot(x="Month", y="Count", data=Colition_weather_df, ci=None)

     plt.ylabel("Daily Incidents",fontsize=20)
     plt.xlabel("Month",fontsize=20)
     plt.title("Daily Incidents per Month",fontsize=24)
```

1.3.8 Average visibility (km) per day (PointPlot)

```
[ ]: fig,ax = plt.subplots(figsize=(15,7))
sns.set(style="darkgrid")
g = sns.pointplot(x="index", y="Visibility (km)", data=Colition_weather_df,␣
 ↪ci=None)

ax.xaxis.set_major_formatter(plt.NullFormatter())

plt.ylabel("Visibility (km)",fontsize=20)
plt.xlabel("Jan         Feb         Mar         Apr         May         Jun        ␣
 ↪Jul        Aug        Sep        Oct        Nov        Dec",fontsize=20)
plt.title("Visibility (km)  per Day",fontsize=24)
```

1.3.9 Average visibility (km) per month (PointPlot)

```
[ ]: fig,ax = plt.subplots(figsize=(15,7))
sns.set(style="darkgrid")
g = sns.pointplot(x="Month", y="Visibility (km)", data=Colition_weather_df,␣
 ↪ci=None )

plt.ylabel("Average Vsisibility (km)",fontsize=20)
plt.xlabel("Month",fontsize=20)
plt.title("Average Visibility (km) per Month",fontsize=24)
```

```
[ ]: #Needed to calculate min and max of visibility data, this will be used to help␣
 ↪determine bin sizes for following figures
maximum = Colition_weather_df['Visibility (km)'].max()
minimum = Colition_weather_df['Visibility (km)'].min()
#print(maximum, minimum) #<-- visual QC
```

1.3.10 Daily incidents vs visibility (km) (PointPlot)

```
[ ]: #create plot of Daily Incidents vs. Average Visibility
Colition_weather_df= Colition_weather_df.sort_values(by=['Visibility (km)'])

labels = ['0 to 2km','2 to 4km','4 to 6km','6 to 8km','8 to 10km','10 to␣
 ↪15km','15 to 20km','20 to 25km','25 to 30km','30 to 35km','35 to 40km','40␣
 ↪to 45km','45 to 50km','50 to 55km']
bins = [0,2,4,6,8,10,15,20,25,30,35,40,45,50,55]

Colition_weather_df['visibility_binned']= pd.
 ↪cut(Colition_weather_df['Visibility (km)'], bins=bins, labels=labels)

fig,ax = plt.subplots(figsize=(15,7))
sns.set(style="darkgrid")
```

```
g = sns.pointplot(x="visibility_binned", y="Count", data=Colition_weather_df,␣
↪ci=None).set_title('Road Incidents vs. Road Visiblity (km)')

plt.ylabel("Daily Incidents",fontsize=20)
plt.xlabel("Average Visibility (km)",fontsize=20)
plt.title("Daily Incidents vs Visibility (km)",fontsize=24)
```

[ ]:
```
#add traffic incidents (but first need to make columns match the weather data␣
↪column formats)
incidents_df = pd.read_csv('..\..\CSV_files\Traffic_Incidents.csv')
incidents_df.head(10)
incidents_df['data']='TrafficIncidents'
#rename columns so they match other dataframes
incidents_df=incidents_df.rename(columns={'Latitude':'latitude','Longitude':
↪'longitude'})
incidents_df=pd.DataFrame(incidents_df)
incidents_df['Time'] = incidents_df['START_DT'].str[10:]
#incidents_df.head() #<-- visual QC
```

[ ]:
```
#split the Time from the hours in the Incident Data. We need to make the␣
↪columns match the weather data so we can combined them
import numpy as np
incidents_df['Time'].str[10:]
incidents_df['Time'].str[:3]
incidents_df['night/day'] = incidents_df['Time'].str[10:]
incidents_df['Hour'] = incidents_df['Time'].str[:3]
incidents_df['Hour'] = incidents_df['Hour'].astype(int)
#incidents_df.head()#<-- visual QC
```

[ ]:
```
#need to change PM/AM time into 24hour clock time, so that we can plot them␣
↪easier on a graph on the X-axis
#create a column for the addition to the current time: 0 for AM's, and +12 for␣
↪PM times:
incidents_df["temp"] = incidents_df["night/day"].map(lambda x: '0' if "AM" in x␣
↪else '12' if "PM" in x else "")
incidents_df['temp'] = incidents_df['temp'].astype(int) #change type to int, so␣
↪we can add columns together
#incidents_df.head() #<-- visual QC
```

[ ]:
```
incidents_df['24HourClock']=incidents_df['Hour']+incidents_df['temp'] #add␣
↪columns to get 24hour clock time
#incidents_df.head()#<-- visual QC
```

1.3.11 Total incidents in 2018, for each hour of the day (Histogram)

[ ]:
```
#Finaly, plot total incidents vs. time of day
bins = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24]

fig,ax = plt.subplots(figsize=(15,7))
sns.distplot(incidents_df['24HourClock'],kde=False , bins = bins,␣
↪hist_kws={"rwidth":0.8,'edgecolor':'black', 'alpha':1.0} )
plt.ylabel("Total Incidents",fontsize=20)
plt.xlabel("Time of Day (24hour clock)",fontsize=20)
plt.xticks([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24])
plt.title("Total Incidents in 2018, For Each Hour of the Day",fontsize=24)
```

1.4 stats.ipynb (computing grid data file)

1.4.1 Libraries and functions

[ ]:
```
import numpy as np
import pandas as pd
import folium
import geopandas as gpd
from shapely.geometry import Polygon
from shapely.geometry import box
```

1.4.2 Creating 10x10 grid

[ ]:
```
df_city_boundary_layer = pd.read_csv('..\..\CSV_files\City_Boundary_Layer.csv')
```

[ ]:
```
string = df_city_boundary_layer["the_geom"].values[0]
string_stripped= string.replace("POLYGON","").replace("(","").replace(")","").
↪replace(",","")
string_split = string_stripped.split()
```

[ ]:
```
list_long = []
list_lat = []

#appends latitudes and longitudes based on long/lat/long/lat... pattern
for i in range(len(string_split)):
    if i % 2 == 0:
        list_long.append(float(string_split[i]))
    else:
        list_lat.append(float(string_split[i]))

min_long = min(list_long)
max_long = max(list_long)
min_lat = min(list_lat)
max_lat = max(list_lat)
```

[ ]:
```
coord_box = box(min_long,min_lat,max_long,max_lat)
geo = gpd.GeoSeries([coord_box]).__geo_interface__
```

[ ]:
```
map_osm = folium.Map(location=[min_lat,min_long], zoom_start=10)
folium.GeoJson(geo).add_to(map_osm)
```

[ ]:
```
map_osm
```

[ ]:
```
long_ten_split = np.linspace(min_long,max_long,num = 11)
lat_ten_split = np.linspace(min_lat,max_lat,num = 11)
```

[ ]:
```
grid_array = []

for y in range(len(lat_ten_split)-1):
    for x in range(len(long_ten_split)-1):
        bot_left = [long_ten_split[x],lat_ten_split[y]]
        bot_right = [long_ten_split[x+1],lat_ten_split[y]]
        top_left = [long_ten_split[x],lat_ten_split[y+1]]
        top_right = [long_ten_split[x+1],lat_ten_split[y+1]]
        grid_array.append([bot_left,bot_right,top_left,top_right])

        coord_box = box(bot_left[0],bot_left[1],top_right[0],top_right[1])
        geo = gpd.GeoSeries([coord_box]).__geo_interface__
        folium.GeoJson(geo).add_to(map_osm)

map_osm
```

1.4.3 Calculations for each 1x1 grid

[ ]:
```
# This method deals with csv files containing multistring longitude/latitude

# Iterate through each line of the csv file. Parse the multistring from the csv␣
↪file to extract the values below for each row and store each value in the␣
↪corresponding list:
# minimum longitude
# minimum latitude
# maximum longitude
# maximum latitude

# Check if a coordinate is within each 1x1 grid.

# Return a list of matching speed limit or volume for each 1x1 grid, depending␣
↪on the parameters passed.


def compute_average_with_multistring(fileName, columnName, stringName,␣
↪columnName2, grid_array):

    df = pd.read_csv(fileName)

    list_long = []
    list_lat = []

    min_long = []
    max_long = []
    min_lat = []
    max_lat = []

    for index, row in df.iterrows():
        # parsing "multiline" column to extract longitude/latitude
        message1 = df[columnName].values[index]
        message1 = message1.replace(stringName, '').replace('(', '').
↪replace(',', '').replace(')', '')
        message1 = message1.split()

        # the multiline string follows longitude, latitude, longitude, latitude.
↪..... pattern
        for i in range(len(message1)):
            if i % 2 == 0:
                list_long.append(float(message1[i]))
            else:
                list_lat.append(float(message1[i]))

        # finding min/max longitude/latitude for each row
```

```
        min_long.append(min(list_long))
        max_long.append(max(list_long))
        min_lat.append(min(list_lat))
        max_lat.append(max(list_lat))

    # consist of 100 lists corresponding to the 100 grids
    # each individual list stores the extracted speed limit if the coordinate␣
↪is within the grid
    resultList = [[] for _ in range(100)]

    for index, row in df.iterrows():
        for i in range(len(grid_array)):

            # To reduce search time:
            # 1. Check if (minimum longitude, minimum latitude) is greater than␣
↪the top right coordinate of the 1x1 grid.
            # 2. Check if (maximum longitude, maximum latitude) is smaller than␣
↪the bottom left coordinate of the 1x1 grid.
            # If the above condition meets, none of the coordinates from the␣
↪current row of the multiline column would lie within the current 1x1 grid.

            # For each 1x1 grid:
            # grid_array[i][3][0] = longitude of top right coordinate
            # grid_array[i][3][1] = latitude of top right coordinate
            # grid_array[i][0][0] = longitude of bottom left coordinate
            # grid_array[i][0][1] = latitude of bottom left coordinate

            if((min_long[index] > grid_array[i][3][0] and min_lat[index] >␣
↪grid_array[i][3][1]) or (max_long[index] < grid_array[i][0][0] and␣
↪max_lat[index] < grid_array[i][0][1])):
                break

            # If the above condition does not meet, there is the possibility of␣
↪having coordinates lie within the current 1x1 grid. We would need to␣
↪continue our search.
            else:
                # parsing "multiline" column to extract longitude/latitude
                message1 = df[columnName].values[index]
                message1 = message1.replace(stringName, '').replace('(', '').
↪replace(',', '').replace(')', '')
                message1 = message1.split()

                # converting to float type
                for x in range(len(message1)):
                    message1[x] = float(message1[x])
```

```
                # If one of the coordinate from the multistring meets the␣
↪condition, we break outside the loop and check the next row.
                # This means that we only consider each road once within each␣
↪grid.
                # For example, if a road appears 5 times within 1 grid, we only␣
↪count it once.
                for j in range(0, len(message1), 2):
                    if((grid_array[i][0][0] <= message1[j]) and (message1[j] <=␣
↪grid_array[i][3][0]) and (grid_array[i][0][1] <= message1[j+1]) and␣
↪(message1[j+1] <= grid_array[i][3][1])):
                        resultList[i].append(df[columnName2].values[index])
                        break
                    else:
                        continue
    return resultList
```

[ ]:
```
# This method deals with csv files containing only 1 value for longitude/
↪latitude

# Iterate through each line of the csv file.

# Check if a coordinate is within each 1x1 grid.

# Return a list of results for each 1x1 grid.

def compute_result(df, columnLong, columnLat, grid_array):

    resultList = [[] for _ in range(100)]

    for index, row in df.iterrows():
        for i in range(len(grid_array)):
            if((grid_array[i][0][0] <= df[columnLong][index]) and␣
↪(df[columnLong][index] <= grid_array[i][3][0]) and (grid_array[i][0][1] <=␣
↪df[columnLat][index]) and (df[columnLat][index] <= grid_array[i][3][1])):
                resultList[i].append(1)
            else:
                continue

    return resultList
```

1.4.4 Computing average speed limit for each area/grid

```
[ ]: # convert to dataframe
     df_average_speed_limit = pd.DataFrame(compute_average_with_multistring('..\..
      ↪\CSV_files\Speed_Limits.csv', 'multiline', 'MULTILINESTRING', 'SPEED',␣
      ↪grid_array))
     # compute average
     df_average_speed_limit['Average Speed Limit'] = df_average_speed_limit.
      ↪mean(axis=1)
     df_average_speed_limit = df_average_speed_limit[['Average Speed Limit']].copy()
```

1.4.5 Computing average traffic volume for each area/grid

```
[ ]: # convert to dataframe
     df_average_traffic_volumes = pd.DataFrame(compute_average_with_multistring('..\.
      ↪.\CSV_files\Traffic_Volumes_for_2018.csv', 'multilinestring',␣
      ↪'MULTILINESTRING', 'VOLUME', grid_array))
     # compute average
     df_average_traffic_volumes['Average Traffic Volume'] =␣
      ↪df_average_traffic_volumes.mean(axis=1)
     df_average_traffic_volumes = df_average_traffic_volumes[['Average Traffic␣
      ↪Volume']].copy()
```

1.4.6 Computing traffic cameras for each area/grid

```
[ ]: df_traffic_cameras = pd.read_csv('..\..\CSV_files\Traffic_Camera_Locations.csv')
     # convert to dataframe
     df_traffic_cameras = pd.DataFrame(compute_result(df_traffic_cameras,␣
      ↪'longitude', 'latitude', grid_array))
     # compute sum
     df_traffic_cameras['Traffic Cameras'] = df_traffic_cameras.sum(axis=1)
     df_traffic_cameras = df_traffic_cameras[['Traffic Cameras']].copy()
```

1.4.7 Computing traffic signals for each area/grid

```
[ ]: df_traffic_signals = pd.read_csv('..\..\CSV_files\Traffic_Signals.csv')
     # convert to dataframe
     df_traffic_signals = pd.DataFrame(compute_result(df_traffic_signals,␣
      ↪'longitude', 'latitude', grid_array))
     # compute sum
     df_traffic_signals['Traffic Signals'] = df_traffic_signals.sum(axis=1)
     df_traffic_signals = df_traffic_signals[['Traffic Signals']].copy()
```

1.4.8 Computing traffic signs for each area/grid

[ ]:
```
df_traffic_signs = pd.read_csv('..\..\CSV_files\Traffic_Signs.csv')

# extracting relevant signs
df_traffic_signs = df_traffic_signs.loc[(df_traffic_signs['BLADE_TYPE'] ==␣
↪'Regulatory') | (df_traffic_signs['BLADE_TYPE'] == 'Warning') |␣
↪(df_traffic_signs['BLADE_TYPE'] == 'Stop') | (df_traffic_signs['BLADE_TYPE']␣
↪== 'Playground') | (df_traffic_signs['BLADE_TYPE'] == 'Yield') |␣
↪(df_traffic_signs['BLADE_TYPE'] == 'Pedestrian') |␣
↪(df_traffic_signs['BLADE_TYPE'] == 'Disabled Parking') |␣
↪(df_traffic_signs['BLADE_TYPE'] == 'Speed') |␣
↪(df_traffic_signs['BLADE_TYPE'] == 'Bicycle / Pathway') |␣
↪(df_traffic_signs['BLADE_TYPE'] == 'School')]

# reset index
df_traffic_signs = df_traffic_signs.reset_index(drop=True)
```

[ ]:
```
signs_per_grid = [[] for _ in range(100)]

for index, row in df_traffic_signs.iterrows():
    for i in range(len(grid_array)):
            message1 = df_traffic_signs['POINT'].values[index]
            message1 = message1.replace('POINT', '').replace('(', '').
↪replace(')', '')
            message1 = message1.split()

            for x in range(2):
                message1[x] = float(message1[x])

            if((grid_array[i][0][0] <= message1[0]) and (message1[0] <=␣
↪grid_array[i][3][0]) and (grid_array[i][0][1] <= message1[1]) and␣
↪(message1[1] <= grid_array[i][3][1])):
                signs_per_grid[i].append(1)
                break
            else:
                continue

df_traffic_signs = pd.DataFrame(signs_per_grid)
# convert to dataframe
df_traffic_signs['Traffic Signs'] = df_traffic_signs.sum(axis=1)
# compute sum
df_traffic_signs = df_traffic_signs[['Traffic Signs']].copy()
```

1.4.9 Computing traffic incidents for each area/grid

[ ]:
```
df_traffic_incidents = pd.read_csv('..\..\CSV_files\Traffic_Incidents.csv')
# extract 2018 incidents
df_traffic_incidents = df_traffic_incidents[df_traffic_incidents['id'].str.
 ↪startswith(str(2018))]

# convert to dataframe
df_traffic_incidents = pd.DataFrame(compute_result(df_traffic_incidents,␣
 ↪'Longitude', 'Latitude', grid_array))
# compute sum
df_traffic_incidents['Traffic Incidents'] = df_traffic_incidents.sum(axis=1)
df_traffic_incidents = df_traffic_incidents[['Traffic Incidents']].copy()
```

1.4.10 Combining results into 1 dataframe

[ ]:
```
# index column corresponds to the 100 grids:
# from bottom left (grid 0) to top right (grid 99)
df = pd.concat([df_average_speed_limit, df_average_traffic_volumes,␣
 ↪df_traffic_cameras, df_traffic_signals, df_traffic_signs,␣
 ↪df_traffic_incidents], axis=1).reindex(df_average_speed_limit.index)
df
```

1.5 graphs_Ivan.ipynb (figure creation from grid data file)

1.5.1 Libraries and functions

[ ]:
```
%matplotlib inline

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

1.5.2 Read combined data csv (containing grid data)

[ ]:
```
df = pd.read_csv('..\..\CSV_files\Combined_Data.csv')
```

1.5.3 Average speed limit vs. incident/volume ratio

[ ]:
```
sns.set(style="white", rc={"lines.linewidth": 3})
fig, ax1 = plt.subplots(figsize=(20,20))
ax2 = ax1.twinx()

sns.barplot(x='Grid', y='Average Speed Limit', data=df, color='cyan', ax=ax1)
sns.lineplot(x='Grid', y='Incidents/Volume Ratio', color='r', data=df, ax=ax2)

plt.locator_params(axis='x', nbins=20)

plt.show()
sns.set()
```

1.5.4 Traffic signals vs. incident/volume ratio

[ ]:
```
sns.set(style="white", rc={"lines.linewidth": 3})
fig, ax1 = plt.subplots(figsize=(20,20))
ax2 = ax1.twinx()

sns.barplot(x='Grid', y='Traffic Signals', data=df, color='cyan', ax=ax1)
sns.lineplot(x='Grid', y='Incidents/Volume Ratio', color='r', data=df, ax=ax2)

plt.locator_params(axis='x', nbins=20)

plt.show()
sns.set()
```

1.5.5 Traffic cameras vs. incident/volume ratio

[ ]:
```
sns.set(style="white", rc={"lines.linewidth": 3})
fig, ax1 = plt.subplots(figsize=(20,20))
ax2 = ax1.twinx()

sns.barplot(x='Grid', y='Traffic Cameras', data=df, color='cyan', ax=ax1)
sns.lineplot(x='Grid', y='Incidents/Volume Ratio', color='r', data=df, ax=ax2)

plt.locator_params(axis='x', nbins=20)

plt.show()
sns.set()
```

1.5.6 Traffic signs vs. incident/volume ratio

[ ]:
```
sns.set(style="white", rc={"lines.linewidth": 3})
fig, ax1 = plt.subplots(figsize=(20,20))
ax2 = ax1.twinx()

sns.barplot(x='Grid', y='Traffic Signs', data=df, color='cyan', ax=ax1)
sns.lineplot(x='Grid', y='Incidents/Volume Ratio', color='r', data=df, ax=ax2)

plt.locator_params(axis='x', nbins=20)

plt.show()
sns.set()
```

1.6 spearman.ipynb (spearman calculating file)

1.6.1 Libraries and functions

[ ]:
```
import pandas as pd
```

[ ]:
```
#creates a list of rankings from a given list
def ranking(input):
    #saves the location of the items in the original list
    value_index_pair = list(zip(input,range(len(input))))

    #sorts the list by the value
    value_index_pair_sorted = sorted(value_index_pair)

    ranks = [0]*len(input)
    #ranks the items in the list and stores it in the original location
    for i, item in enumerate(value_index_pair_sorted):
        #print(i,item)
        ranks[item[1]] = i+1

    return ranks
```

[ ]:
```
#computes the spearman coefficient given two ranking lists
def compute_spearman_ranks(x_ranks,y_ranks):
    n = len(x_ranks)

    d_square = []

    #d^2 for each row of data ranks input
    for x,y in zip(x_ranks,y_ranks):
        diff = x - y
        d_square.append(diff**2)

    #sum of d^2 terms
    sum_d_square = sum(d_square)

    #calculate spearman coefficient
    rs = 1 - (6*sum_d_square/(n*(n**2-1)))
    return rs
```

1.6.2 Read csv files

[ ]:
```
#read combined csv file (grid data)
combined_df = pd.read_csv("..\..\CSV_files\Combined_Data.csv")

#drop nan values in traffic volume
combined_dropna_df = combined_df[combined_df["Average Traffic Volume"].notna()]
combined_dropna_df["Incidents/Volume Ratio"] = combined_dropna_df["Total␣
↪Incidents"]/combined_dropna_df["Average Traffic Volume"]

#read weather data with incident count
weather_df = pd.read_csv("..\..\CSV_files\WeatherIncidents.csv")
```

1.6.3 Calculate and output spearman coefficients

[ ]:
```
#calculate incident ranking normalizing for traffic volume (for grid data) (x␣
↪value)
incident_volume = list(combined_dropna_df["Incidents/Volume Ratio"])
incident_volume_rank = ranking(incident_volume)

#calculate incident ranking normalizing for daily rate(for weather data) (x␣
↪value)
incident_count = list(weather_df["Count"])
incident_count_rank = ranking(incident_count)

#calculate camera number ranking (y value) and spearman coefficient
cameras = list(combined_dropna_df["Total Cameras"])
cameras_rank = ranking(cameras)
cameras_spear = compute_spearman_ranks(cameras_rank, incident_volume_rank)

#calculate signals number ranking (y value) and spearman coefficient
signals = list(combined_dropna_df["Total Signals"])
signals_rank = ranking(signals)
signals_spear = compute_spearman_ranks(signals_rank, incident_volume_rank)

#calculate signs number ranking (y value) and spearman coefficient
signs = list(combined_dropna_df["Total Signs"])
signs_rank = ranking(signs)
signs_spear = compute_spearman_ranks(signs_rank, incident_volume_rank)

#remove nan values in average speed
combined_dropna_df = combined_dropna_df[combined_dropna_df["Average Speed␣
↪Limit"].notna()]
incident_volume = list(combined_dropna_df["Incidents/Volume Ratio"])
incident_volume_rank = ranking(incident_volume)

#calculate speed number ranking (y value) and spearman coefficient
```

```
speed = list(combined_dropna_df["Average Speed Limit"])
speed_rank = ranking(speed)
speed_spear = compute_spearman_ranks(speed_rank, incident_volume_rank)

#calculate visibility ranking (y value) and spearman coefficient
visib = list(weather_df["Visibility (km)"])
visib_rank = ranking(visib)
visib_spear = compute_spearman_ranks(visib_rank, incident_volume_rank)

#calculate temperature ranking (y value) and spearman coefficient
temp = list(weather_df["Mean Temp (C)"])
temp_rank = ranking(temp)
temp_spear = compute_spearman_ranks(temp_rank, incident_volume_rank)
```

[ ]:
```
#grid data spearman coefficients
print("Grid data")
print("Spearman correlation against incident/volume")
print("Cameras:",cameras_spear)
print("Signals:",signals_spear)
print("Signs:",signs_spear)
print("Speed:",speed_spear, "\n")

#weather data spearman coefficients
print("Weather data")
print("Spearman correlation against average daily incidents")
print("Visibility:",visib_spear)
print("Temperature:",temp_spear)
```

1.7 mapping_all.ipynb (mapping file)

1.7.1 Libraries and functions

[ ]:
```
#import required libraries
import numpy as np
import pandas as pd
import folium
import geopandas as gpd
from folium import plugins
from folium.plugins import HeatMap
from folium.plugins import FloatImage
from shapely.geometry import box
from shapely.geometry import MultiLineString
import branca
from colour import Color
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
from PIL import Image
```

[ ]:
```
#CODE BORROWED FROM https://nbviewer.jupyter.org/gist/BibMartin/
 ↪f153aa957ddc5fadc64929abdee9ff2e
from branca.element import MacroElement

from jinja2 import Template

class BindColormap(MacroElement):
    """Binds a colormap to a given layer.

    Parameters
    ----------
    colormap : branca.colormap.ColorMap
        The colormap to bind.
    """
    def __init__(self, layer, colormap):
        super(BindColormap, self).__init__()
        self.layer = layer
        self.colormap = colormap
        self._template = Template(u"""
        {% macro script(this, kwargs) %}
            {{this.colormap.get_name()}}.svg[0][0].style.display = 'block';
            {{this._parent.get_name()}}.on('overlayadd', function (eventLayer) {
                if (eventLayer.layer == {{this.layer.get_name()}}) {
                    {{this.colormap.get_name()}}.svg[0][0].style.display =␣
 ↪'block';
                }});
            {{this._parent.get_name()}}.on('overlayremove', function␣
 ↪(eventLayer) {
```

```
                if (eventLayer.layer == {{this.layer.get_name()}}) {
                    {{this.colormap.get_name()}}.svg[0][0].style.display =␣
↪'none';
                }});
        {% endmacro %}
        """)  # noqa
```

[ ]:
```
#function for creating heatmap given lat, long and weight data
def create_heatmap(list_lat, list_long, map_osm, heat_name, list_weight = [], r␣
↪= 20):
    #create feature layer
    feature = folium.map.FeatureGroup(name = heat_name, overlay = True, show =␣
↪False)

    #if weights are given
    if (len(list_weight) > 0):
        #create heatmap
        plugins.HeatMap(zip(list_lat, list_long, list_weight), radius = r,␣
↪gradient = {0.25: "blue", 0.4: "lightblue", 0.6: "lime", 0.8: "yellow", 1.0:␣
↪"red"}).add_to(feature)

        #create legend
        legend = create_legend(list_weight)

        #add feature to map and bind feature and legend together
        map_osm.add_child(feature)
        map_osm.add_child(legend)
        map_osm.add_child(BindColormap(feature,legend))

    #otherwise don't use any weights
    else:
        #create heatmap
        plugins.HeatMap(zip(list_lat, list_long), radius = r, gradient = {0.25:␣
↪"blue", 0.4: "lightblue", 0.6: "lime", 0.8: "yellow", 1.0: "red"}).
↪add_to(feature)

        #create legend
        legend = create_legend()

        #add feature to map and bind feature and legend together
        map_osm.add_child(feature)
        map_osm.add_child(legend)
        map_osm.add_child(BindColormap(feature,legend))
```

[ ]:
```
#function for creating heatmap given lat, long and weight data (specifically␣
↪for incidents)
```

```python
def create_heatmap_incidents(list_lat, list_long, map_osm, heat_name, list_weight = [], r = 20):
    #create feature layer
    feature = folium.map.FeatureGroup(name = heat_name, overlay = True, show = False)

    #if weights are given
    if (len(list_weight) > 0):
        #create heatmap
        #inferno color scheme (reversed)
        plugins.HeatMap(zip(list_lat, list_long), radius = r, gradient = {0.05: "#e4fa15", 0.142857143: "#f6bd27", 0.285714286: "#ee8949",  0.428571429: "#d85b69", 0.571428571: "#ba2f8a", 0.714285714: "#9200a6", 0.857142857: "#6200a4", 1.0: "#2f0087"}).add_to(feature)

        #create legend
        legend = create_legend_incidents(list_weight)

        #add feature to map and bind feature and legend together
        map_osm.add_child(feature)
        map_osm.add_child(legend)
        map_osm.add_child(BindColormap(feature,legend))

    #otherwise don't use any weights
    else:
        #create heatmap
        #inferno color scheme (reversed)
        plugins.HeatMap(zip(list_lat, list_long), radius = r, gradient = {0.05: "#e4fa15", 0.142857143: "#f6bd27", 0.285714286: "#ee8949",  0.428571429: "#d85b69", 0.571428571: "#ba2f8a", 0.714285714: "#9200a6", 0.857142857: "#6200a4", 1.0: "#2f0087"}).add_to(feature)

        #create legend
        legend = create_legend_incidents()

        #add feature to map and bind feature and legend together
        map_osm.add_child(feature)
        map_osm.add_child(legend)
        map_osm.add_child(BindColormap(feature,legend))
```

[ ]:

```python
#function for generating legend (returns a linear interpolated color gradient)
def create_legend(list_weight = []):
    #colors and their respective index values
    colors = ["blue", "lightblue", "lime", "yellow", "red"]
    index = [0.25, 0.4, 0.6, 0.8, 1.0]

    #if weights are given, scale the index
```

```
    if (len(list_weight) > 0):
        #scale index with weight values
        max_weight = max(list_weight)
        min_weight = min(list_weight)
        diff_weight = max_weight - min_weight
        index = [(i*diff_weight) + min_weight for i in index]

        #create a linear interpolation of colors
        colormap = branca.colormap.LinearColormap(colors = colors, index =␣
↪index, vmin = min_weight, vmax = max_weight)

    #else, use standard index
    else:
        colormap = branca.colormap.LinearColormap(colors = colors, index =␣
↪index)

    return colormap
```

[ ]:
```
#function for generating legend (returns a linear interpolated color gradient)␣
↪(specifically for incidents)
def create_legend_incidents(list_weight = []):
    #colors and their respective index values
    # colors = ["#2f0087", "#6200a4", "#9200a6", "#ba2f8a", "#d85b69",␣
↪"#ee8949", "#f6bd27", "#e4fa15"]
    colors = ["#e4fa15", "#f6bd27", "#ee8949", "#d85b69", "#ba2f8a", "#9200a6",␣
↪"#6200a4", "#2f0087"]
    index = [0.05, 0.142857143, 0.285714286, 0.428571429, 0.571428571, 0.
↪714285714, 0.857142857, 1.0]

    #if weights are given, scale the index
    if (len(list_weight) > 0):
        #scale index with weight values
        max_weight = max(list_weight)
        min_weight = min(list_weight)
        diff_weight = max_weight - min_weight
        index = [(i*diff_weight) + min_weight for i in index]

        #create a linear interpolation of colors
        colormap = branca.colormap.LinearColormap(colors = colors, index =␣
↪index, vmin = min_weight, vmax = max_weight)

    #else, use standard index
    else:
        colormap = branca.colormap.LinearColormap(colors = colors, index =␣
↪index)
```

```
    return colormap
```

[ ]:
```
#function for creating grid markers
def create_markers(map_osm, grid_arr):
    #create feature layer
    feature = folium.map.FeatureGroup(name = "grid markers", overlay = True,␣
 ↪show = True)

    #open combined csv file and replace nan with string NaN
    combined_df = pd.read_csv("..\..\CSV_files\Combined_Data.csv")
    combined_df.fillna("NaN", inplace = True)

    #iterate through rows
    for index, row in combined_df.iterrows():
        #find middle point of grid
        grid = grid_arr[row["Grid"]]
        long_coord = (grid[0][0] + grid[3][0]) / 2
        lat_coord = (grid[0][1] + grid[3][1]) / 2
        column_names = list(combined_df.columns.values)

        #create string of row data
        text = ""
        for col_name in column_names:
            text += "<br>" + col_name + ": " + str(row[col_name]) + "<br>"

        #add to feature layer
        iframe = folium.IFrame(text, width=220, height=310)
        popup = folium.Popup(iframe)
        folium.Marker(location=[lat_coord, long_coord], popup=popup).
 ↪add_to(feature)

    #add to map
    map_osm.add_child(feature)
```

[ ]:
```
#function for adding Calgary 10x10 grid array onto the map
def create_grid_array(grid_array, map_osm):
    #create feature layer
    feature = folium.map.FeatureGroup(name = "grid array", overlay = True, show␣
 ↪= True)

    #iterate through each grid
    for grid in grid_array:
        #find the 4 corners of the grid
        coord_box = box(grid[0][0],grid[0][1],grid[3][0],grid[3][1])

        #create box geojson and add to feature layer
        geo = gpd.GeoSeries([coord_box]).__geo_interface__
```

```
        folium.GeoJson(geo).add_to(folium.FeatureGroup(name='grid array')).
↪add_to(feature)

    #add feature layer to map
    map_osm.add_child(feature)
```

1.7.2 Creating 10x10 grid

```
[ ]: #Read city boundary layer
df = pd.read_csv("..\..\CSV_files\City_Boundary_layer.csv")
# df
```

```
[ ]: #parse the coordinates
string = df["the_geom"].values[0]
string_stripped= string.replace("POLYGON","").replace("(","").replace(")","").
↪replace(",","")
string_split = string_stripped.split()
```

```
[ ]: list_long = []
list_lat = []

#appends latitudes and longitudes based on long/lat/long/lat... pattern
for i in range(len(string_split)):
    if i % 2 == 0:
        list_long.append(float(string_split[i]))
    else:
        list_lat.append(float(string_split[i]))

#find the min and max values of the boundary
min_long = min(list_long)
max_long = max(list_long)
min_lat = min(list_lat)
max_lat = max(list_lat)
# print(min_long)
# print(max_long)
```

```
[ ]: #creation of grid array for a 10x10 grid
long_ten_split = np.linspace(min_long,max_long,num = 11)
lat_ten_split = np.linspace(min_lat,max_lat,num = 11)

grid_array = []
for y in range(len(lat_ten_split)-1):
    for x in range(len(long_ten_split)-1):
        bot_left = [long_ten_split[x],lat_ten_split[y]]
        bot_right = [long_ten_split[x+1],lat_ten_split[y]]
        top_left = [long_ten_split[x],lat_ten_split[y+1]]
        top_right = [long_ten_split[x+1],lat_ten_split[y+1]]
        grid_array.append([bot_left,bot_right,top_left,top_right])
```

1.7.3 Traffic volume heat map

[ ]:
```
#read traffic csv
traffic_df = pd.read_csv("..\..\CSV_files\Traffic_Volumes_for_2018.csv")
# traffic_df
```

[ ]:
```
#parse data for lat, longs and weight (volume)
traffic_data = []

#parse multilinestring
for string,volume in zip(traffic_df['multilinestring'],traffic_df["VOLUME"]):
    string_stripped= string.replace("MULTILINESTRING","").replace("(","").
 ↪replace(")","").replace(",","")
    string_split = string_stripped.split()
    float_split = [float(i) for i in string_split]

    for i in range(int(len(float_split)/2)):
        traffic_data.append([float_split[i*2+1],float_split[i*2],float(volume)])


list_lat = [i[0] for i in traffic_data]
list_long = [i[1] for i in traffic_data]
list_weight = [i[2] for i in traffic_data]
```

[ ]:
```
#create map
map_osm = folium.Map(location = [50.913577283979,-114.
 ↪073657541927],control_scale = True, zoom_start=10)

#add heatmap
create_heatmap(list_lat, list_long, map_osm, heat_name = "volume", list_weight␣
 ↪= list_weight)

# create_grid_array(grid_array, map_osm)
# create_markers(map_osm, grid_array)
# folium.LayerControl().add_to(map_osm)

# map_osm.save("traffic_heat_map_grid_marker.html")
# map_osm
```

1.7.4 Traffic camera heat map

[ ]:
```
#read traffic camera locations csv
camera_df = pd.read_csv("..\..\CSV_files\Traffic_Camera_Locations.csv")
# camera_df
```

[ ]:
```
#parse data for lats and longs
list_long = [i for i in camera_df["longitude"]]
list_lat = [i for i in camera_df["latitude"]]
```

[ ]:
```
# map_osm = folium.Map(location = [50.913577283979,-114.
 ↪073657541927],control_scale = True, zoom_start=10)

#add heatmap
create_heatmap(list_lat, list_long, map_osm, heat_name = "camera", r = 25)
# create_grid_array(grid_array, map_osm)
# create_markers(map_osm, grid_array)
# folium.LayerControl().add_to(map_osm)

# map_osm.save("traffic_heat_map_grid_marker.html")
# map_osm
```

1.7.5 Traffic signals heat map

[ ]:
```
#read traffic signals csv
signals_df = pd.read_csv("..\..\CSV_files\Traffic_Signals.csv")
# signals_df
```

[ ]:
```
#parse data for lats and longs
list_long = [i for i in signals_df["longitude"]]
list_lat = [i for i in signals_df["latitude"]]
```

[ ]:
```
#create map
# map_osm = folium.Map(location = [50.913577283979,-114.
 ↪073657541927],control_scale = True, zoom_start=10)

#add heatmap
create_heatmap(list_lat, list_long, map_osm, heat_name = "signals", r = 17)
# create_grid_array(grid_array, map_osm)
# create_markers(map_osm, grid_array)
# folium.LayerControl().add_to(map_osm)

# map_osm.save("traffic_heat_map_grid_marker.html")
# map_osm
```

1.7.6 Traffic signs heat map

[ ]:
```
#read traffic signs csv
signs_df = pd.read_csv("..\..\CSV_files\Traffic_Signs.csv")

#drop nan and 0 values in sign count and sign type
signs_df.dropna(subset=["SGN_COUNT_NO"], how='all', inplace=True)
signs_df.dropna(subset=["BLADE_TYPE"], how='all', inplace=True)
signs_df = signs_df[signs_df.SGN_COUNT_NO != 0]

#list of "irrelevant signs" (parking signs, info signs etc.)
signs_nan = ['Timed Parking','Park Plus','Parking Restrictions','Street␣
↪Name','Snow Route','Guide / Information','Loading Zone','Residential␣
↪Parking', 'Overhead Guide']

#remove irrelevant signs
signs_df = signs_df.loc[~signs_df["BLADE_TYPE"].isin(signs_nan)]

# signs_df
```

[ ]:
```
#parse data for lats, longs and weight (sign count)
list_long = []
list_lat = []
list_weight = [i for i in signs_df["SGN_COUNT_NO"]]

#parse POINT string
for row in signs_df["POINT"]:
    string_stripped = row.replace("POINT", "").replace("(", "").replace(")", "")
    string_split = string_stripped.split()

    list_long.append(float(string_split[0]))
    list_lat.append(float(string_split[1]))
```

[ ]:
```
#create map
# map_osm = folium.Map(location = [50.9135777283979,-114.
↪073657541927],control_scale = True, zoom_start=10)

#add heatmap
create_heatmap(list_lat, list_long, map_osm, heat_name = "signs (all)",␣
↪list_weight = list_weight, r = 15)
# create_grid_array(grid_array, map_osm)
# create_markers(map_osm, grid_array)
# folium.LayerControl().add_to(map_osm)

# map_osm.save("traffic_heat_map_grid_marker.html")
# map_osm
```

1.7.7 Traffic incidents

[ ]:
```
#read traffic signs csv
incidents_df = pd.read_csv("..\..\CSV_files\Traffic_Incidents.csv")

#parse for 2018 data
incidents_df = incidents_df[incidents_df.START_DT.str.contains("2018")]
# incidents_df
```

[ ]:
```
#parse data for lats and longs
list_long = [i for i in incidents_df["Longitude"]]
list_lat = [i for i in incidents_df["Latitude"]]
```

[ ]:
```
#create map
# map_osm = folium.Map(location = [50.913577283979,-114.
↪073657541927],control_scale = True, zoom_start=10)

#add heatmap
create_heatmap(list_lat, list_long, map_osm, heat_name = "incidents (rainbow)",␣
↪r = 20)
create_heatmap_incidents(list_lat, list_long, map_osm, heat_name = "incidents",␣
↪r = 20)
# create_grid_array(grid_array, map_osm)
# create_markers(map_osm, grid_array)
# folium.LayerControl().add_to(map_osm)

# map_osm.save("traffic_heat_map_grid_marker.html")
# map_osm
```

1.7.8 Traffic signs (stop, warning yield) specific heat map

```
[ ]: #read traffic signs csv
signs_df = pd.read_csv("..\..\CSV_files\Traffic_Signs.csv")

#drop nan and 0 values in sign count and sign type
signs_df.dropna(subset=["SGN_COUNT_NO"], how='all', inplace=True)
signs_df.dropna(subset=["BLADE_TYPE"], how='all', inplace=True)
signs_df = signs_df[signs_df.SGN_COUNT_NO != 0]

#list of "irrelevant signs" (parking signs, info signs etc.)
# signs_nan = ['Timed Parking','Park Plus','Parking Restrictions','Street␣
↪Name','Snow Route','Guide / Information','Loading Zone','Disabled␣
↪Parking','Residential Parking', 'Overhead Guide']

#remove irrelevant signs
signs_df = signs_df.loc[signs_df["BLADE_TYPE"].isin(["Stop","Warning","Yield"])]

# signs_df
```

```
[ ]: #parse data for lats, longs and weight (sign count)
list_long = []
list_lat = []
list_weight = [i for i in signs_df["SGN_COUNT_NO"]]

#parse POINT string
for row in signs_df["POINT"]:
    string_stripped = row.replace("POINT", "").replace("(", "").replace(")", "")
    string_split = string_stripped.split()

    list_long.append(float(string_split[0]))
    list_lat.append(float(string_split[1]))
```

```
[ ]: #create map
# map_osm = folium.Map(location = [50.913577283979,-114.
↪073657541927],control_scale = True, zoom_start=10)

#add heatmap
create_heatmap(list_lat, list_long, map_osm, heat_name = "signs (stop, warning,␣
↪yield)", list_weight = list_weight, r = 15)
# create_grid_array(grid_array, map_osm)
# create_markers(map_osm, grid_array)
# folium.LayerControl().add_to(map_osm)

# map_osm.save("traffic_heat_map_grid_marker.html")
# map_osm
```

1.7.9 Traffic signs (speed) specific heat map

[ ]:
```
#read traffic signs csv
signs_df = pd.read_csv("..\..\CSV_files\Traffic_Signs.csv")

#drop nan and 0 values in sign count and sign type
signs_df.dropna(subset=["SGN_COUNT_NO"], how='all', inplace=True)
signs_df.dropna(subset=["BLADE_TYPE"], how='all', inplace=True)
signs_df = signs_df[signs_df.SGN_COUNT_NO != 0]

#list of "irrelevant signs" (parking signs, info signs etc.)
# signs_nan = ['Timed Parking','Park Plus','Parking Restrictions','Street␣
↪Name','Snow Route','Guide / Information','Loading Zone','Disabled␣
↪Parking','Residential Parking', 'Overhead Guide']

#remove irrelevant signs
signs_df = signs_df.loc[signs_df["BLADE_TYPE"].isin(["Speed"])]

# signs_df
```

[ ]:
```
#parse data for lats, longs and weight (sign count)
list_long = []
list_lat = []
list_weight = [i for i in signs_df["SGN_COUNT_NO"]]

#parse POINT string
for row in signs_df["POINT"]:
    string_stripped = row.replace("POINT", "").replace("(", "").replace(")", "")
    string_split = string_stripped.split()

    list_long.append(float(string_split[0]))
    list_lat.append(float(string_split[1]))
```

[ ]:
```
#create map
# map_osm = folium.Map(location = [50.913577283979,-114.
↪073657541927],control_scale = True, zoom_start=10)

#add heatmap
create_heatmap(list_lat, list_long, map_osm, heat_name = "signs (speed)",␣
↪list_weight = list_weight, r = 20)
# create_grid_array(grid_array, map_osm)
# create_markers(map_osm, grid_array)
# folium.LayerControl().add_to(map_osm)

# map_osm.save("traffic_heat_map_grid_marker.html")
# map_osm
```

1.7.10 Traffic signs (pedestrians and bicycle pathways) specific heat map

[ ]:
```
#read traffic signs csv
signs_df = pd.read_csv("..\..\CSV_files\Traffic_Signs.csv")

#drop nan and 0 values in sign count and sign type
signs_df.dropna(subset=["SGN_COUNT_NO"], how='all', inplace=True)
signs_df.dropna(subset=["BLADE_TYPE"], how='all', inplace=True)
signs_df = signs_df[signs_df.SGN_COUNT_NO != 0]

#list of "irrelevant signs" (parking signs, info signs etc.)
# signs_nan = ['Timed Parking','Park Plus','Parking Restrictions','Street␣
↪Name','Snow Route','Guide / Information','Loading Zone','Disabled␣
↪Parking','Residential Parking', 'Overhead Guide']

#remove irrelevant signs
signs_df = signs_df.loc[signs_df["BLADE_TYPE"].isin(['Pedestrian','Bicycle /␣
↪Pathway'])]

# signs_df
```

[ ]:
```
#parse data for lats, longs and weight (sign count)
list_long = []
list_lat = []
list_weight = [i for i in signs_df["SGN_COUNT_NO"]]

#parse POINT string
for row in signs_df["POINT"]:
    string_stripped = row.replace("POINT", "").replace("(", "").replace(")", "")
    string_split = string_stripped.split()

    list_long.append(float(string_split[0]))
    list_lat.append(float(string_split[1]))
```

[ ]:
```
#create map
# map_osm = folium.Map(location = [50.913577283979,-114.
↪073657541927],control_scale = True, zoom_start=10)

#add heatmap
create_heatmap(list_lat, list_long, map_osm, heat_name = "signs (pedestrian,␣
↪bicycle)", list_weight = list_weight, r = 15)
# create_grid_array(grid_array, map_osm)
# create_markers(map_osm, grid_array)
# folium.LayerControl().add_to(map_osm)

# map_osm.save("traffic_heat_map_grid_marker.html")
# map_osm
```

1.7.11 Traffic signs (playground and schools) specific heat map

[ ]:
```
#read traffic signs csv
signs_df = pd.read_csv("..\..\CSV_files\Traffic_Signs.csv")

#drop nan and 0 values in sign count and sign type
signs_df.dropna(subset=["SGN_COUNT_NO"], how='all', inplace=True)
signs_df.dropna(subset=["BLADE_TYPE"], how='all', inplace=True)
signs_df = signs_df[signs_df.SGN_COUNT_NO != 0]

#list of "irrelevant signs" (parking signs, info signs etc.)
# signs_nan = ['Timed Parking','Park Plus','Parking Restrictions','Street␣
↪Name','Snow Route','Guide / Information','Loading Zone','Disabled␣
↪Parking','Residential Parking', 'Overhead Guide']

#remove irrelevant signs
signs_df = signs_df.loc[signs_df["BLADE_TYPE"].isin(['Playground','School'])]

# signs_df
```

[ ]:
```
#parse data for lats, longs and weight (sign count)
list_long = []
list_lat = []
list_weight = [i for i in signs_df["SGN_COUNT_NO"]]

#parse POINT string
for row in signs_df["POINT"]:
    string_stripped = row.replace("POINT", "").replace("(", "").replace(")", "")
    string_split = string_stripped.split()

    list_long.append(float(string_split[0]))
    list_lat.append(float(string_split[1]))
```

[ ]:
```
#create map
# map_osm = folium.Map(location = [50.913577283979,-114.
↪073657541927],control_scale = True, zoom_start=10)

#add heatmap
create_heatmap(list_lat, list_long, map_osm, heat_name = "signs (playground,␣
↪school)", list_weight = list_weight, r = 15)
# create_grid_array(grid_array, map_osm)
# create_markers(map_osm, grid_array)
# folium.LayerControl().add_to(map_osm)

# map_osm.save("layered_heat_map_grid_marker.html")
# map_osm
```

1.7.12 Speed limit road colors

[ ]:
```
#import traffic Speed Limits
speed_df = pd.read_csv('..\..\CSV_files\Speed_Limits.csv')

#take relevant columns
speed_df = speed_df[["SPEED","multiline"]]
# speed_df
```

[ ]:
```
#speed list
list_speed = [i for i in speed_df["SPEED"]]
```

[ ]:
```
#parse multiline string for coordinate list
list_coord_row =[]

for row in speed_df["multiline"]:

    #strip MULTILINESTRING and (
    string = row.replace("MULTILINESTRING","").replace("(","")

    #split on every line
    string_split = string.split("),")

    list_coord = []
    for string in string_split:
        #find coordinate pairs
        pairs = string.split(",")
        list_pair = []

        for pair in pairs:
            #convert each coordinate into a float
            pair_strip = pair.strip().replace(")","")
            pair_split = pair_strip.split()
            float_split = [float(i) for i in pair_split]

            list_pair.append(tuple(float_split))
        list_coord.append([list_pair])
    list_coord_row.append(list_coord)
```

[ ]:
```
#create color gradient
yellow = Color("yellow")
colors = list(yellow.range_to(Color("red"),10))
colors = [str(i) for i in colors]
colors.insert(2,'#ffd400')
colors.insert(4,'#ffb800',)

keys = [i for i in range(20,111,10)]
```

```
keys.append(35)
keys.append(45)
keys.sort()

#create dictionary for speeds to corresponding color
colors_dict = dict(zip(keys,colors))
```

[ ]:
```
#creating legend using pyplot
fig, ax = plt.subplots()
fig.set_size_inches(5, 10.5)
handles = [mpatches.Patch(color=colors_dict[x], label=x) for x in colors_dict.
 ↪keys()]
ax.legend(handles = handles, fontsize = 30, loc = "center", title = "Speed␣
 ↪limit (km/h)", title_fontsize = "20")
fig.gca().set_axis_off()
fig.savefig("..\..\HTML_files\legend.png")

#suppress output
plt.close()
```

[ ]:
```
feature_speed = folium.map.FeatureGroup(name = "speed limits", overlay = True,␣
 ↪show = True)

#add legend image to html
FloatImage("legend.png", bottom=-10, left=85).add_to(feature_speed)

#add roads and their corresponding color based on the speed limit
for row,speed in zip(list_coord_row, list_speed):
    for line in row:
        for coordinate in line:
            style = {'fillColor': colors_dict[speed], 'color':␣
 ↪colors_dict[speed]}
            lines = MultiLineString([coordinate])
            geo = gpd.GeoSeries([lines]).__geo_interface__
            folium.GeoJson(geo, style_function=lambda x, fillColor =␣
 ↪style["fillColor"], color = style["color"]: {"fillColor": fillColor, "color":
 ↪ color}).add_to(feature_speed)
```

[ ]:
```
#create map
# map_osm = folium.Map(location = [50.9135772839779,-114.
 ↪073657541927],control_scale = True, zoom_start=10)

#add heatmap
# create_heatmap(list_lat, list_long, map_osm, heat_name = "signs (playground,␣
 ↪school)", list_weight = list_weight, r = 15)
```

```python
#create grid array and markers feature layers
map_osm.add_child(feature_speed)
create_grid_array(grid_array, map_osm)
create_markers(map_osm, grid_array)

# #add layer control
folium.LayerControl(position = "bottomleft").add_to(map_osm)

map_osm.save("..\..\HTML_files\layered_heat_map_grid_marker.html");
# map_osm
```