Home-Assignment

November 4, 2023

[1]:
```
!pip install gdown
```

```
Requirement already satisfied: gdown in c:\users\malir\anaconda3\lib\site-
packages (4.7.1)
Requirement already satisfied: beautifulsoup4 in
c:\users\malir\anaconda3\lib\site-packages (from gdown) (4.11.1)
Requirement already satisfied: requests[socks] in
c:\users\malir\anaconda3\lib\site-packages (from gdown) (2.28.1)
Requirement already satisfied: filelock in c:\users\malir\anaconda3\lib\site-
packages (from gdown) (3.6.0)
Requirement already satisfied: six in c:\users\malir\anaconda3\lib\site-packages
(from gdown) (1.16.0)
Requirement already satisfied: tqdm in c:\users\malir\anaconda3\lib\site-
packages (from gdown) (4.64.1)
Requirement already satisfied: soupsieve>1.2 in
c:\users\malir\anaconda3\lib\site-packages (from beautifulsoup4->gdown) (2.3.1)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in
c:\users\malir\anaconda3\lib\site-packages (from requests[socks]->gdown)
(1.26.11)
Requirement already satisfied: idna<4,>=2.5 in
c:\users\malir\anaconda3\lib\site-packages (from requests[socks]->gdown) (3.3)
Requirement already satisfied: charset-normalizer<3,>=2 in
c:\users\malir\anaconda3\lib\site-packages (from requests[socks]->gdown) (2.0.4)
Requirement already satisfied: certifi>=2017.4.17 in
c:\users\malir\anaconda3\lib\site-packages (from requests[socks]->gdown)
(2022.9.14)
Requirement already satisfied: PySocks!=1.5.7,>=1.5.6 in
c:\users\malir\anaconda3\lib\site-packages (from requests[socks]->gdown) (1.7.1)
Requirement already satisfied: colorama in c:\users\malir\anaconda3\lib\site-
packages (from tqdm->gdown) (0.4.5)
```

1 Importing libraries

[2]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import warnings
```

```
import gdown
import zipfile
```

2 Loading,unzipping,saving and reading the CSV file

[3]:
```
warnings.filterwarnings('ignore')

# Exctracting the id from url
file_id = '1QnpOWeRAlycJLGsr4ghyB-TzS4cQpaLl'
url = f'https://drive.google.com/uc?id={file_id}'
output = 'Mo-Alrz.zip'

# Download the file from Google Drive
gdown.download(url, output, quiet=False)

# Unzipping and saving in extracted directory
extracted = "Mohammad Alirezaee-Home Assignment"
with zipfile.ZipFile(output, 'r') as zip_ref:
    zip_ref.extractall(extracted)

csv_file_path = f"{extracted}/
 ↪Conditions_Contributing_to_COVID-19_Deaths__by_State_and_Age__Provisional_2020-2023.
 ↪csv"

# Loading data into a pandas dataframe
df = pd.read_csv(csv_file_path)
```

```
Downloading…
From: https://drive.google.com/uc?id=1QnpOWeRAlycJLGsr4ghyB-TzS4cQpaLl
To: C:\Users\malir\Mo-Alrz.zip
100%|
 | 3.66M/3.66M [00:00<00:00, 17.9MB/s]
```

3 Inspecting the data and its characteristics

First step to build a machine learning model is the Exploratory analysis, but in this case because we have certain steps to take and because the focus of this course is on building data models, i will stick to those particular steps, just before that, i will try to become familiar with the dataset husing .head(), .describe() and .info() methods.

[4]:
```
df.head(10)
```

[4]:

	Data As Of	Start Date	End Date	Group	Year	Month	State	\
0	06/25/2023	01/01/2020	06/24/2023	By Total	NaN	NaN	United States	
1	06/25/2023	01/01/2020	06/24/2023	By Total	NaN	NaN	United States	
2	06/25/2023	01/01/2020	06/24/2023	By Total	NaN	NaN	United States	
3	06/25/2023	01/01/2020	06/24/2023	By Total	NaN	NaN	United States	

```
4  06/25/2023  01/01/2020  06/24/2023  By Total   NaN    NaN  United States
5  06/25/2023  01/01/2020  06/24/2023  By Total   NaN    NaN  United States
6  06/25/2023  01/01/2020  06/24/2023  By Total   NaN    NaN  United States
7  06/25/2023  01/01/2020  06/24/2023  By Total   NaN    NaN  United States
8  06/25/2023  01/01/2020  06/24/2023  By Total   NaN    NaN  United States
9  06/25/2023  01/01/2020  06/24/2023  By Total   NaN    NaN  United States

        Condition Group                Condition ICD10_codes   Age Group  \
0  Respiratory diseases  Influenza and pneumonia     J09-J18        0-24
1  Respiratory diseases  Influenza and pneumonia     J09-J18       25-34
2  Respiratory diseases  Influenza and pneumonia     J09-J18       35-44
3  Respiratory diseases  Influenza and pneumonia     J09-J18       45-54
4  Respiratory diseases  Influenza and pneumonia     J09-J18       55-64
5  Respiratory diseases  Influenza and pneumonia     J09-J18       65-74
6  Respiratory diseases  Influenza and pneumonia     J09-J18       75-84
7  Respiratory diseases  Influenza and pneumonia     J09-J18         85+
8  Respiratory diseases  Influenza and pneumonia     J09-J18  Not stated
9  Respiratory diseases  Influenza and pneumonia     J09-J18    All Ages

   COVID-19 Deaths  Number of Mentions Flag
0           1554.0              1630.0  NaN
1           5775.0              5998.0  NaN
2          15026.0             15643.0  NaN
3          37335.0             38794.0  NaN
4          82382.0             85404.0  NaN
5         128349.0            132400.0  NaN
6         137362.0            140693.0  NaN
7         119833.0            121695.0  NaN
8             12.0                12.0  NaN
9         527628.0            542269.0  NaN
```

[5]:
```
df.describe()
```

[5]:
```
                Year          Month  COVID-19 Deaths  Number of Mentions
count  571320.000000  521640.000000     4.111710e+05        4.167610e+05
mean     2021.304348       6.071429     1.266096e+02        1.362334e+02
std         1.039850       3.425349     3.052289e+03        3.279466e+03
min      2020.000000       1.000000     0.000000e+00        0.000000e+00
25%      2020.000000       3.000000     0.000000e+00        0.000000e+00
50%      2021.000000       6.000000     0.000000e+00        0.000000e+00
75%      2022.000000       9.000000     1.900000e+01        2.100000e+01
max      2023.000000      12.000000     1.135624e+06        1.135624e+06
```

[6]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 583740 entries, 0 to 583739
Data columns (total 14 columns):
```

```
 #   Column              Non-Null Count   Dtype
---  ------              --------------   -----
 0   Data As Of          583740 non-null  object
 1   Start Date          583740 non-null  object
 2   End Date            583740 non-null  object
 3   Group               583740 non-null  object
 4   Year                571320 non-null  float64
 5   Month               521640 non-null  float64
 6   State               583740 non-null  object
 7   Condition Group     583740 non-null  object
 8   Condition           583740 non-null  object
 9   ICD10_codes         583740 non-null  object
 10  Age Group           583740 non-null  object
 11  COVID-19 Deaths     411171 non-null  float64
 12  Number of Mentions  416761 non-null  float64
 13  Flag                172569 non-null  object
dtypes: float64(4), object(10)
memory usage: 62.4+ MB
```

4 Data Preparation

4.1 Restrict the dataset , Start Date = "01/01/2020" , End Date = "06/24/2023"

[7]:
```
df["Start Date"] = pd.to_datetime(df["Start Date"], format='%m/%d/%Y')
df["End Date"] = pd.to_datetime(df["End Date"], format='%m/%d/%Y')

start_date_filtered = pd.to_datetime('01/01/2020', format='%m/%d/%Y')
end_date_filtered = pd.to_datetime('06/24/2023', format='%m/%d/%Y')

filtered_df = df[(df["Start Date"] == start_date_filtered) & (df["End Date"] ==␣
 ↪end_date_filtered)]
```

Checking the restricted dates

[8]:
```
filtered_df["Start Date"].unique()
```

[8]:
```
array(['2020-01-01T00:00:00.000000000'], dtype='datetime64[ns]')
```

[9]:
```
filtered_df["End Date"].unique()
```

[9]:
```
array(['2023-06-24T00:00:00.000000000'], dtype='datetime64[ns]')
```

4.2 Remove rows not belonging to specific age groups

First we check the age groups values and we see 'Not stated' and 'All Ages' are not specific groups so we get rid of those

[10]:
```
filtered_df["Age Group"].unique()
```

[10]:
```
array(['0-24', '25-34', '35-44', '45-54', '55-64', '65-74', '75-84',
       '85+', 'Not stated', 'All Ages'], dtype=object)
```

[11]:
```
for i in ["Not stated","All Ages"]:
  filtered_df = filtered_df[filtered_df['Age Group'] != i]
```

Then we check the age groups again to make sure

[12]:
```
filtered_df['Age Group'].unique()
```

[12]:
```
array(['0-24', '25-34', '35-44', '45-54', '55-64', '65-74', '75-84',
       '85+'], dtype=object)
```

5 Remove rows with the United States as State

[13]:
```
filtered_df = filtered_df[filtered_df["State"] != "United States"]
```

Double check

[14]:
```
filtered_df["State"].unique()
```

[14]:
```
array(['Alabama', 'Alaska', 'Arizona', 'Arkansas', 'California',
       'Colorado', 'Connecticut', 'Delaware', 'District of Columbia',
       'Florida', 'Georgia', 'Hawaii', 'Idaho', 'Illinois', 'Indiana',
       'Iowa', 'Kansas', 'Kentucky', 'Louisiana', 'Maine', 'Maryland',
       'Massachusetts', 'Michigan', 'Minnesota', 'Mississippi',
       'Missouri', 'Montana', 'Nebraska', 'Nevada', 'New Hampshire',
       'New Jersey', 'New Mexico', 'New York', 'New York City',
       'North Carolina', 'North Dakota', 'Ohio', 'Oklahoma', 'Oregon',
       'Pennsylvania', 'Rhode Island', 'South Carolina', 'South Dakota',
       'Tennessee', 'Texas', 'Utah', 'Vermont', 'Virginia', 'Washington',
       'West Virginia', 'Wisconsin', 'Wyoming', 'Puerto Rico'],
      dtype=object)
```

6 Investigate missing values, and deal with them if you find any

[15]:
```
filtered_df.isnull().sum()
```

[15]:
```
Data As Of              0
Start Date              0
End Date                0
Group                   0
Year                 9752
Month                9752
State                   0
Condition Group         0
Condition               0
ICD10_codes             0
```

```
Age Group                0
COVID-19 Deaths       1646
Number of Mentions    1572
Flag                  8106
dtype: int64
```

[16]:
```
filtered_df["Data As Of"].value_counts()
```

[16]:
```
06/25/2023    9752
Name: Data As Of, dtype: int64
```

[17]:
```
filtered_df["Group"].value_counts()
```

[17]:
```
By Total    9752
Name: Group, dtype: int64
```

[18]:
```
filtered_df["Flag"].value_counts()
```

[18]:
```
One or more data cells have counts between 1-9 and have been suppressed in
accordance with NCHS confidentiality standards.    1646
Name: Flag, dtype: int64
```

We checked the missing values, but before handling those values we would like to get rid of unnecessary columns since it will ease the handling missing values step, so the column "Data As Of" because it has only one value and probably its the date of the day in which,the data file was created,"Year" and "Month" because the whole columns are empty, "Group" because only value is By Total and i dont think it can be of any interest to us,"ICD10_codes" maybe some high level jargon regarding the presentation of different types of viruse,"Flag" again has 8106 empty cells (aproximately 8106/9752 = 83 %) so i will get rid of this as well

[19]:
```
filtered_df.drop(columns=["Data As␣
↪Of","Group","Year","Month","ICD10_codes","Flag"],inplace=True)
filtered_df
```

[19]:

	Start Date	End Date	State	Condition Group	\
230	2020-01-01	2023-06-24	Alabama	Respiratory diseases	
231	2020-01-01	2023-06-24	Alabama	Respiratory diseases	
232	2020-01-01	2023-06-24	Alabama	Respiratory diseases	
233	2020-01-01	2023-06-24	Alabama	Respiratory diseases	
234	2020-01-01	2023-06-24	Alabama	Respiratory diseases	
…	…	…	…	…	
12413	2020-01-01	2023-06-24	Puerto Rico	COVID-19	
12414	2020-01-01	2023-06-24	Puerto Rico	COVID-19	
12415	2020-01-01	2023-06-24	Puerto Rico	COVID-19	
12416	2020-01-01	2023-06-24	Puerto Rico	COVID-19	
12417	2020-01-01	2023-06-24	Puerto Rico	COVID-19	

```
                    Condition Age Group  COVID-19 Deaths  Number of Mentions
230    Influenza and pneumonia      0-24             20.0                20.0
231    Influenza and pneumonia     25-34            103.0               108.0
232    Influenza and pneumonia     35-44            230.0               237.0
233    Influenza and pneumonia     45-54            547.0               564.0
234    Influenza and pneumonia     55-64           1188.0              1224.0
…                          …       …                …                   …
12413                COVID-19     45-54            439.0               439.0
12414                COVID-19     55-64            780.0               780.0
12415                COVID-19     65-74           1238.0              1238.0
12416                COVID-19     75-84           1640.0              1640.0
12417                COVID-19       85+           1713.0              1713.0

[9752 rows x 8 columns]
```

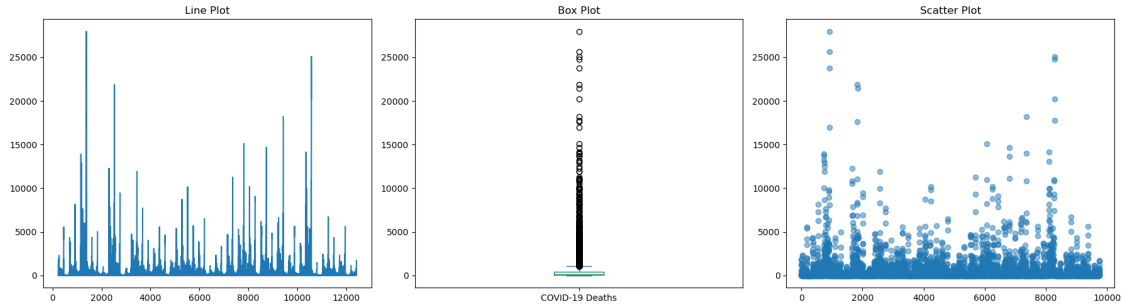
[20]:
```
filtered_df.isnull().sum()
```

[20]:
```
Start Date              0
End Date                0
State                   0
Condition Group         0
Condition               0
Age Group               0
COVID-19 Deaths      1646
Number of Mentions   1572
dtype: int64
```

For the "COVID-19 Deaths" because this column contains the real values that was observed and if we plot the line,scatter, and box plots we see it does not have a normal distribution and because of fluctuations in number of deaths lots of data points treat like outliers, the reason is, during the pandemic we had some peaks which had a normal distribution(increasing,reaching the maximum and then decreasing) even though it wasnt possible to predict the pattern for the next peak and the plot of too many peaks that we have is completely imbalance. So filling missing values can not be a good idea because it will cause bias in our data and model predictions and even if we want to fill this data because of distribution and imbalance can not be filled with mean,median, or even if we want to fill with zero again it wil lead to have a completely biased predcitions and probably the loss function will be too high.

[21]:
```
fig, axes = plt.subplots(1, 3, figsize=(18, 5))

filtered_df["COVID-19 Deaths"].plot.line(ax=axes[0])
axes[0].set_title("Line Plot")

filtered_df["COVID-19 Deaths"].plot.box(ax=axes[1])
axes[1].set_title("Box Plot")

x_values = range(len(filtered_df))
```

```
y_values = filtered_df["COVID-19 Deaths"]
axes[2].scatter(x_values, y_values, marker='o', alpha=0.5)
axes[2].set_title("Scatter Plot")

plt.tight_layout()

plt.show()
```



So i simply drop the missing values (rows) in "COVID-19 Deaths"

[22]:
```
filtered_df.dropna(subset=["COVID-19 Deaths"],inplace=True)
```

[23]:
```
filtered_df.isnull().sum()
```

[23]:
```
Start Date            0
End Date              0
State                 0
Condition Group       0
Condition             0
Age Group             0
COVID-19 Deaths       0
Number of Mentions    0
dtype: int64
```

We can see by dropping those missing values in "COVID-19 Deaths" column, those rows that were containing the missing values in "Number of Mentions" column were removed as well and the reason for that is the high correlation between these two columns in the primary dataset (df), and now we have handled all the missing values in our data set.

[24]:
```
"{:.3f}".format(df["COVID-19 Deaths"].corr(df["Number of Mentions"]))
```

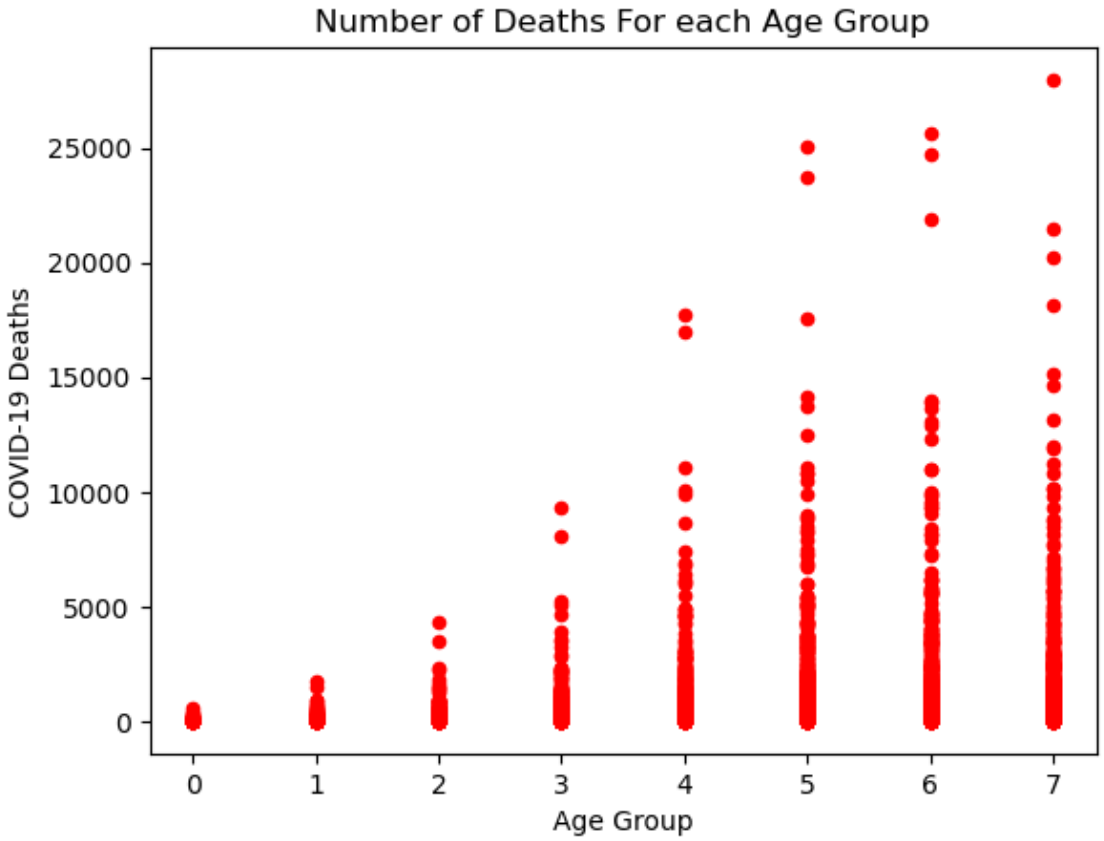
[24]:
```
'0.987'
```

7 Encode age groups as you see fit as numeric columns

[25]:
```
age_group = {'0-24': 0,'25-34': 1,'35-44': 2,'45-54': 3,'55-64': 4,'65-74':␣
↪5,'75-84': 6,'85+': 7}
filtered_df["Age Group"] = filtered_df["Age Group"].map(age_group)
```

[26]:
```
filtered_df[["COVID-19 Deaths","Age Group"]].plot.scatter(y="COVID-19␣
↪Deaths",x="Age Group",c="red",title="Number of Deaths For each Age Group")
```

[26]:
```
<AxesSubplot:title={'center':'Number of Deaths For each Age Group'}, xlabel='Age
Group', ylabel='COVID-19 Deaths'>
```



As we can see there is a clear relationship between age groups and number of deaths, the higher the age group, the higher the number of covid-19 deaths, so we can assume age group as an ordinal variable, this shows that

- For KNN we will one-hot encode as it relies on distance metrics, which are more meaningful when using one-hot encoding for categorical variables
- SVMs are versatile and can handle both one-hot encoded and label encoded data. However, in practice, one-hot encoding is often preferred for SVMs because it avoids the potential

misconception of equal intervals between categories, so we will stick with the one-hot encoded version

- Logistic regression is typically used for binary or multi-class classification problems. When dealing with categorical variables, one-hot encoding is usually better because it prevents the model from making inappropriate assumptions about ordinal relationships, so in this case as well we will stick with the one-hot encoded version
- For algorithms like random forest it makes sense to use label encoded age group, because it can effectively handle ordinal data and they make decisions based on a series of splits in individual decision trees.

So just in case of random forest i will use the label encoded dataset.

8 Create a Broad Condition Group variable, which should be the same as the Condition Group variable for the two most frequent condition groups, but has the value "other" for all other condition groups.

[27]:
```
# Finding two most frequent condition groups
condition_dict = filtered_df["Condition"].value_counts().to_dict()
two_most_frequent = sorted(condition_dict, key=condition_dict.get,␣
 ↪reverse=True)[:2]

# Putting values in new cloumn and changing it into two most frequent and others
condition_dict = {k: k if k in two_most_frequent else "Other" for k in␣
 ↪condition_dict}
filtered_df["Broad Condition Group"] = filtered_df["Condition"].
 ↪map(condition_dict)
```

9 Label-encode the Broad Condition Group variable.

[28]:
```
# Encoding values into 0 1 2
broad_cond_grp = {'COVID-19':0, 'All other conditions and causes (residual)':
 ↪1,"Other":2}
filtered_df["Broad Condition Group"] = filtered_df["Broad Condition Group"].
 ↪map(broad_cond_grp)
```

10 Here we reset the indexes because it's unsorted and untidy and we want those to begin from 0 and end at the number of length of the dataset

[29]:
```
filtered_df = filtered_df.reset_index(drop=True)
```

11 Scaling the "COVID-19 Deaths" column

[30]:
```
print(f"Max:{filtered_df['COVID-19 Deaths'].max():.2f},Min:
↪{filtered_df['COVID-19 Deaths'].min():.2f},Mean:{filtered_df['COVID-19␣
↪Deaths'].mean():.2f},Standard Devation:{filtered_df['COVID-19 Deaths'].std():
↪.2f}")
```

```
Max:27972.00,Min:0.00,Mean:541.72,Standard Devation:1470.82
```

Because "COVID-19 Deaths" is a continuous numerical variable with a significant range but other feature was transformed to 0-7 age groups,and because "COVID-19 Deaths" vary a lot and contains remarkable number of outliers (as we can see it varies from 0 to 27972 with a mean of 541 and STD of 1470) and we are trying to implement some ML algortihms we should apply feature scaling on this column. For a column with these characteristics we will use Standardization to have a mean of 0 and a standard deviation of 1. It assumes that the data follows a normal distribution, but it can work reasonably well for many machine learning algorithms, even if the features are not normally distributed. Standardization is less sensitive to outliers compared to Min-Max scaling.

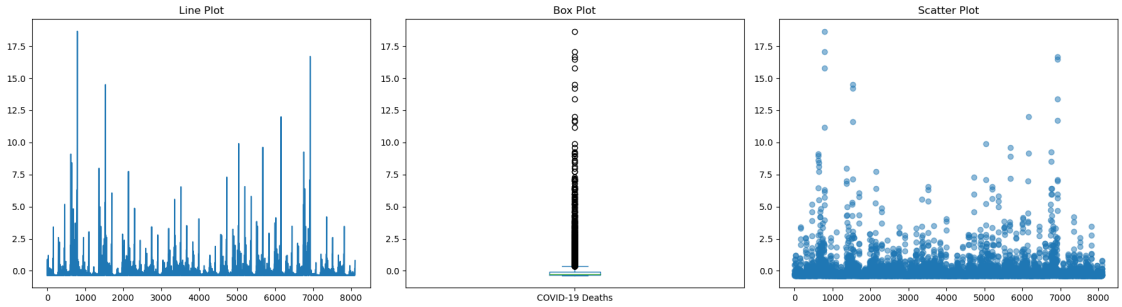
[31]:
```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
filtered_df[["COVID-19 Deaths"]] = scaler.fit_transform(filtered_df[["COVID-19␣
↪Deaths"]])
print(f"Max:{filtered_df['COVID-19 Deaths'].max():.2f},Min:
↪{filtered_df['COVID-19 Deaths'].min():.2f},Mean:{filtered_df['COVID-19␣
↪Deaths'].mean():.2f},Standard Devation:{filtered_df['COVID-19 Deaths'].std():
↪.2f}")
```

```
Max:18.65,Min:-0.37,Mean:0.00,Standard Devation:1.00
```

We should bear in mind that scaling won't change the underlying distribution of the data. It only transforms the values within the same distribution to a new scale or range. The purpose of scaling is to make the values more suitable for certain machine learning algorithms and to ensure that they have similar scales for better model performance. We plot the same line,scatter and box plots for "COVID-19 Deaths" To see that the distribuiton remains the same after scaling.

[32]:
```
fig, axes = plt.subplots(1, 3, figsize=(18, 5))

filtered_df["COVID-19 Deaths"].plot.line(ax=axes[0])
axes[0].set_title("Line Plot")

filtered_df["COVID-19 Deaths"].plot.box(ax=axes[1])
axes[1].set_title("Box Plot")

x_values = range(len(filtered_df))
y_values = filtered_df["COVID-19 Deaths"]
```

```
axes[2].scatter(x_values, y_values, marker='o', alpha=0.5)
axes[2].set_title("Scatter Plot")

plt.tight_layout()

plt.show()
```



12 Keeping Features and Target and removing other unnecessary columns

Retain the columns that are relevant and informative for our analysis or modeling (Features and target) and delete columns that do not provide useful information, are redundant, or are unrelated to our analysis or modeling task. This can include columns with constant values, identifiers, metadata, or columns that we have decided are not relevant to the problem you are solving. Eventually the pre-processed dataset that we are going to work with, will be named final_df.

[33]:
```
label_encoded = filtered_df[["Age Group","COVID-19 Deaths","Broad Condition␣
↪Group"]]

one_hot_encoded = label_encoded.copy()
one_hot_encoded = pd.get_dummies(one_hot_encoded,columns=["Age Group"])
```

13 Implementing classification algorithms and building models

13.1 Train-Test Split

Train-test splitting is a fundamental technique in machine learning. It involves dividing our dataset into two subsets : a training set (usually 70-80 % of the data) and a test set (usually 30-20 % of the data). We fit our data on the training data (train the data), and we test our model (make predictions on target variable) on the test part, and because the test part is not seen by the model the results produced by that will be a good assessment of how well our model performs. A very crucial hyper parameter here is the random state which is used to increase the reproduce-ability and also it helps us to get same output whenever we run this notebook. It is neseccary to go with the same random state untill the end of our tasks.

- The train-validation-test split is a variation of the traditional train-test split that includes an additional validation set and we are going to use. This method is particularly useful when we want to fine-tune hyperparameters and evaluate our model's performance more rigorously. So we will first train our data and make predictions on X_validation (validation dataset) then find the best hyper parameters, and with those hyper parameters predict the X_test data set to get the best results and model. So we will with most common proportion which is 70 % For train, then 30% for temp which is divided into 15 % validation and 15 % test.

[34]:
```
from sklearn.model_selection import train_test_split
X = label_encoded[["Age Group","COVID-19 Deaths"]]
y = label_encoded[["Broad Condition Group"]]

X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3,␣
 ↪random_state=42)
X_validation, X_test, y_validation, y_test = train_test_split(X_temp, y_temp,␣
 ↪test_size=0.5, random_state=42)
```

13.2 K Nearest Neighbor

K-Nearest Neighbors (KNN) is a simple and intuitive supervised machine learning algorithm used for classification and regression tasks (In our case for classification). It's a non-parametric and instance-based learning algorithm, meaning it doesn't make strong assumptions about the underlying data distribution. Instead, it makes predictions based on the proximity (similarity) of data points in a feature space. In other words it is called K Nearest Neighbour because it considers the K closest data points (neighbors) in the training dataset to make predictions.

In implementing these algorithm n_neighbours (K) is our hyper parameter and can significantly impact the performance of the KNN algorithm. A small K like 1 can lead to a noisy model that might overfit the data, while a large K can lead to a smoother decision boundary but might underfit the data. So it is extremely essential that we find the optimum K (Hyperparameter optimization step). There are two main methods for optimizing the K:

- The formula k = sqrt(N)/2 is a common heuristic for choosing the number of neighbors (K) in the K-Nearest Neighbors (KNN) algorithm. However, it's important to remember that this is a rule of thumb and should be used as a starting point rather than a strict rule. The value of K chosen using this formula can be too large for many datasets (like our dataset in which the k value will be 40), leading to a KNN model that is overly complex and potentially overfits the data
- The other thing we know is that when we have two features like our case we need to provide an odd number for K, so we tried k = 1,3,5. The metrics for 5 was higher and more reasonable, i didn't go further (k = 7,9,...) because even if they provide better answer our model will be much more complex and we always want to avoid that and have a simple model that is computationally affordable as well.

[35]:
```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import precision_score, recall_score, f1_score,␣
 ↪accuracy_score, confusion_matrix

scores = []
for i in [1,3,5]:
    knn = KNeighborsClassifier(n_neighbors = i)
    knn.fit(X_train,y_train)
    knn_valid_preds = knn.predict(X_validation)

    knn_valid_precision = precision_score(y_validation, knn_valid_preds,␣
 ↪average='weighted')
    knn_valid_recall = recall_score(y_validation, knn_valid_preds,␣
 ↪average='weighted')
    knn_valid_f1 = f1_score(y_validation, knn_valid_preds, average='weighted')
    knn_valid_acc = accuracy_score(y_validation, knn_valid_preds)
    scores.append(f"for n_neighbors = {i} -> Presicion:{knn_valid_precision:.
 ↪3f},Recall:{knn_valid_recall:.3f},F1:{knn_valid_f1:.3f},Accuracy:
 ↪{knn_valid_acc:.3f}")

for i in scores:
    print(i)
```

```
for n_neighbors = 1 -> Presicion:0.861,Recall:0.843,F1:0.851,Accuracy:0.843
for n_neighbors = 3 -> Presicion:0.858,Recall:0.869,F1:0.864,Accuracy:0.869
for n_neighbors = 5 -> Presicion:0.858,Recall:0.873,F1:0.865,Accuracy:0.873
```

Hyperparameter in these metrics is average, in multiclass classification, it's common to have class imbalances where some classes have significantly more instances than others. The weighted precision considers the class imbalance by assigning a weight to each class based on its relative frequency in the dataset.

Now we train the model with best hyper parametes on the full training set and make predictions on X_test

[44]:
```
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)
knn_preds = knn.predict(X_test)

knn_precision = precision_score(y_test, knn_preds, average='weighted')
knn_recall = recall_score(y_test, knn_preds, average='weighted')
knn_f1 = f1_score(y_test, knn_preds, average='weighted')
knn_acc = accuracy_score(y_test, knn_preds)

# We will create this dictionary and score the matrics of each model in that so␣
 ↪that in the end we have a table for comparing
scores_dict = {"KNN":["{:.4f}".format(knn_precision),"{:.4f}".
 ↪format(knn_recall),
```

```
                        "{:.4f}".format(knn_f1),"{:.4f}".format(knn_acc)]}
```

14 Logistic Regression

Logistic Regression can be extended to handle multiclass classification problems, which involve predicting one of multiple classes, in our case because we have 3 classes in our target variable, The use of hyper parameter class_weight="balanced" in scikit-learn's LogisticRegression class is designed to handle multiclass classification problems, and it will automatically adjust the class weights based on the distribution of classes in your training data. Here because we don't have hyperparameter i will concatenate the validation and test column to have a new 30% test sample.

[45]:
```
from sklearn.linear_model import LogisticRegression

X_test_new = pd.concat([X_validation, X_test], axis=0)
y_test_new = pd.concat([y_validation, y_test], axis=0)

logreg = LogisticRegression(class_weight="balanced")
logreg.fit(X_train,y_train)
logreg_preds = logreg.predict(X_test_new)
```

[46]:
```
logreg_precision = precision_score(y_test_new, logreg_preds, average='weighted')
logreg_recall = recall_score(y_test_new, logreg_preds, average='weighted')
logreg_f1 = f1_score(y_test_new, logreg_preds, average='weighted')
logreg_acc = accuracy_score(y_test_new, logreg_preds)

scores_dict["Logistic Regression"] = ["{:.4f}".format(logreg_precision),"{:.
 ↪4f}".format(logreg_recall),
                                      "{:.4f}".format(logreg_f1),"{:.4f}".
 ↪format(logreg_acc)]
```

Feature Importance

[47]:
```
feature_importance = logreg.coef_[0]
feature_names = X_train.columns
feature_importance_dict = dict(zip(feature_names, feature_importance))

for feature, importance in feature_importance_dict.items():
    print(f'{feature}: {importance:.4f}')
```

```
Age Group: -0.2325
COVID-19 Deaths: 0.8897
```

- an increase in "COVID-19 Deaths" is associated with an increase in the log-odds of being in class 0,1,2
- an increase in "Age Group" is associated with a decrease in the log-odds of being in class 0,1,2

[48]:
```
intercept = logreg.intercept_
intercept
```

[48]:
```
array([ 0.59772184,  0.21391897, -0.81164081])
```

- Intercept for Class 0: The baseline log-odds of being in Class 0 when all feature values are zero, compared to Class 1 and Class 2. In this case, the log-odds are approximately 0.5977 for Class 0.
- Intercept for Class 1: The baseline log-odds of being in Class 1 when all feature values are zero, compared to Class 0 and Class 2. In this case, the log-odds are approximately 0.2139 for Class 1.
- Intercept for Class 2: The baseline log-odds of being in Class 2 when all feature values are zero, compared to Class 0 and Class 1. In this case, the log-odds are approximately -0.8116 for Class 2.

15 SVM : Linear

Linear SVMs can work well with one-hot encoded categorical data. Each one-hot encoded category becomes a binary feature, and SVMs aim to find a hyperplane that best separates the classes in the feature space.

[49]:
```
from sklearn.svm import SVC

svm_linear = SVC(kernel="linear", probability=True)
svm_linear.fit(X_train,y_train)
svm_linear_preds = svm_linear.predict(X_test)
```

[50]:
```
svm_linear_precision = precision_score(y_test, svm_linear_preds,␣
 ↪average='weighted')
svm_linear_recall = recall_score(y_test, svm_linear_preds, average='weighted')
svm_linear_f1 = f1_score(y_test, svm_linear_preds, average='weighted')
svm_linear_accuracy = accuracy_score(y_test, svm_linear_preds)

scores_dict["SVM Linear"] = ["{:.4f}".format(svm_linear_precision),"{:.4f}".
 ↪format(svm_linear_recall),"{:.4f}".format(svm_linear_f1),"{:.4f}".
 ↪format(svm_linear_accuracy)]
scores_dict
```

[50]:
```
{'KNN': ['0.8762', '0.8947', '0.8846', '0.8947'],
 'Logistic Regression': ['0.8817', '0.6953', '0.7648', '0.6953'],
 'SVM Linear': ['0.8424', '0.9021', '0.8601', '0.9021']}
```

16 SVM : Polynomial

Polynomial SVM extends the capabilities of the Linear SVM to handle non-linearly separable data by using a polynomial kernel function. The polynomial kernel maps the data into a higher-dimensional space where a linear decision boundary can separate the classes. The degree of the polynomial kernel controls the complexity of the

decision boundary. Higher degrees can capture more complex patterns but may also risk overfitting. Polynomial SVM is effective for datasets where a non-linear decision boundary is more appropriate. It's suitable for problems where the relationship between features and the target is not strictly linear.

Using label encoding instead of one-hot encoding can be more efficient when working with SVM polynominals, especially when we're dealing with a dataset that has a large number of categorical features or a dataset with high dimensionality, One hot encoding in our case because of 8 different columns in age group will be time consuming and computationally not affordable.

[52]:
```
for i in [3,4,5]:
    svm_poly = SVC(kernel="poly", degree=i)
    svm_poly.fit(X_train,y_train)
    svm_poly_preds = svm_poly.predict(X_validation)

    svm_poly_precision = precision_score(y_validation, svm_poly_preds,␣
 ↪average='weighted')
    svm_poly_recall = recall_score(y_validation, svm_poly_preds,␣
 ↪average='weighted')
    svm_poly_f1 = f1_score(y_validation, svm_poly_preds, average='weighted')
    svm_poly_accuracy = accuracy_score(y_validation, svm_poly_preds)
    scores_poly.append(f"for degree = {i} -> Presicion:{svm_poly_precision:.
 ↪3f},Recall:{svm_poly_recall:.3f},F1:{svm_poly_f1:.3f},Accuracy:
 ↪{svm_poly_accuracy:.3f}")

for i in scores_poly:
    print(i)
```

```
for degree = 3 -> Presicion:0.826,Recall:0.909,F1:0.865,Accuracy:0.909
for degree = 4 -> Presicion:0.826,Recall:0.909,F1:0.865,Accuracy:0.909
for degree = 5 -> Presicion:0.874,Recall:0.910,F1:0.869,Accuracy:0.910
```

[54]:
```
svm_poly = SVC(kernel="poly", degree=5)
svm_poly.fit(X_train,y_train)
svm_poly_preds = svm_poly.predict(X_test)

svm_poly_precision = precision_score(y_validation, svm_poly_preds,␣
 ↪average='weighted')
svm_poly_recall = recall_score(y_validation, svm_poly_preds, average='weighted')
svm_poly_f1 = f1_score(y_validation, svm_poly_preds, average='weighted')
svm_poly_accuracy = accuracy_score(y_validation, svm_poly_preds)

scores_dict[f"SVM Poly {5}"] = ["{:.4f}".format(svm_poly_precision),"{:.4f}".
 ↪format(svm_poly_recall),"{:.4f}".format(svm_poly_f1),"{:.4f}".
 ↪format(svm_poly_accuracy)]
```

```
{'KNN': ['0.8762', '0.8947', '0.8846', '0.8947'], 'Logistic Regression':
['0.8817', '0.6953', '0.7648', '0.6953'], 'SVM Linear': ['0.8424', '0.9021',
```

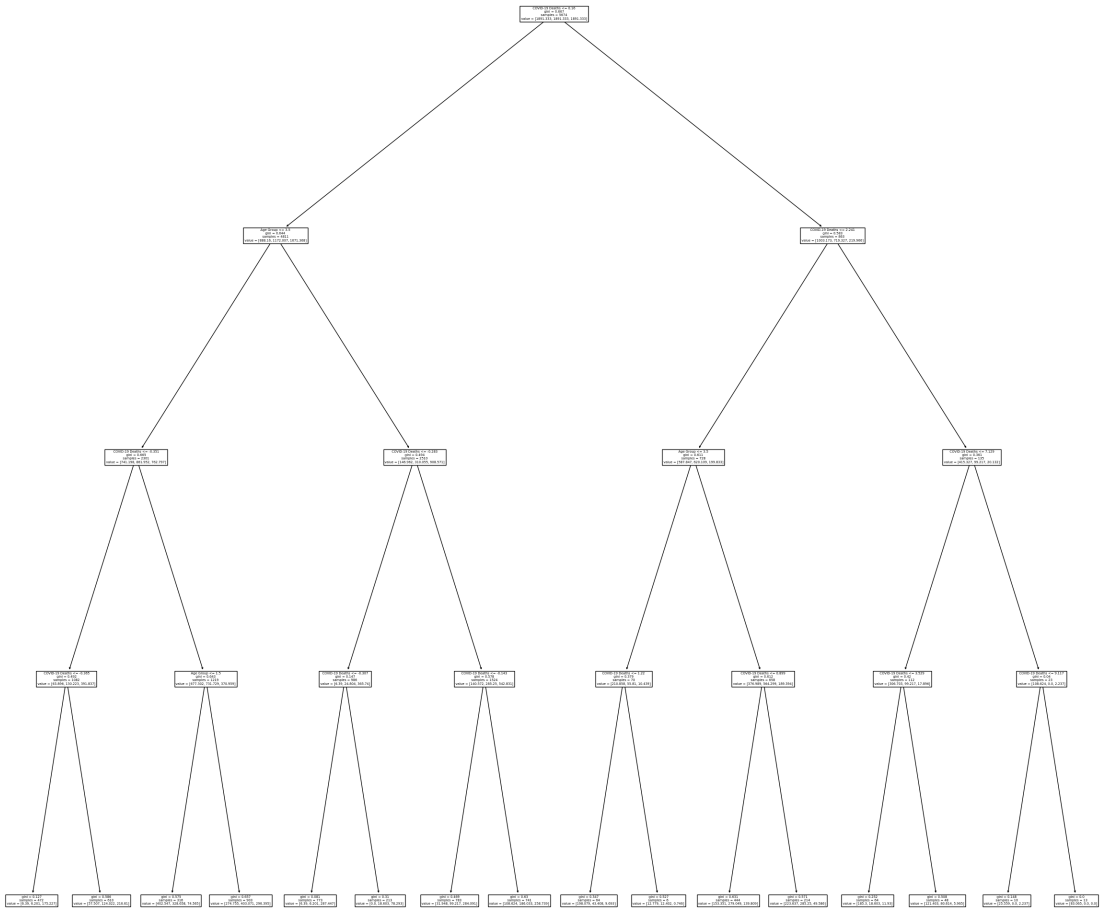
```
'0.8601', '0.9021'], 'SVM Poly 5': ['0.8255', '0.9054', '0.8636', '0.9054']}
```

17 Decision Tree

[55]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn import tree

dec_tree = DecisionTreeClassifier(max_depth=4, min_samples_leaf=5,␣
 ↪random_state=42, class_weight="balanced")
dec_tree.fit(X_train,y_train)
dec_tree_preds = dec_tree.predict(X_test)
```

[57]:
```
dec_tree_precision = precision_score(y_test, dec_tree_preds, average='weighted')
dec_tree_recall = recall_score(y_test, dec_tree_preds, average='weighted')
dec_tree_f1 = f1_score(y_test, dec_tree_preds, average='weighted')
dec_tree_accuracy = f1_score(y_test, dec_tree_preds, average='weighted')

scores_dict["Decision Tree"] =␣
 ↪[dec_tree_precision,dec_tree_recall,dec_tree_f1,dec_tree_accuracy]
```

[58]:
```
plt.figure(figsize=(30,30))
tree.plot_tree(dec_tree, feature_names=dec_tree.feature_names_in_)
plt.show()
```



[60]:
```
scores_df = pd.DataFrame(scores_dict,index=['Precision', 'Recall',␣
 ↪'F1','Accuracy'])
scores_df
```

[60]:

	KNN	Logistic Regression	SVM Linear	SVM Poly 5	Decision Tree
Precision	0.8762	0.8817	0.8424	0.8255	0.893091
Recall	0.8947	0.6953	0.9021	0.9054	0.642270
F1	0.8846	0.7648	0.8601	0.8636	0.733500
Accuracy	0.8947	0.6953	0.9021	0.9054	0.733500

[ ]: