import pandas as pd

def filter\_country\_data(data, country\_name):

country\_data = data[data['location'] == country\_name]

return country\_data

# Load the dataset

covid\_data = pd.read\_csv('owid-covid-data.csv')

def rolling\_mean(series, window\_size):

rolling\_means = []

for i in range(len(series)):

start\_index = max(0, i - window\_size + 1)

end\_index = i + 1

rolling\_mean = series[start\_index:end\_index].mean()

rolling\_means.append(rolling\_mean)

return rolling\_means

sg\_data['total\_cases\_per\_million'].ewm(span=7, adjust=False).mean()

gives more weight to recent data points. This can reduce lag while still smoothing the data.

A black and white text

Description automatically generated

Yes, there are alternative ways to convert a column to datetime format in Pandas. Here are a few other methods you can use:

1. \*\*pd.to\_datetime() with errors parameter\*\*: If your date column contains some invalid dates or different date formats, you can use the `errors` parameter to handle them. For example:

```python

sg\_data['date'] = pd.to\_datetime(sg\_data['date'], errors='coerce')

```

Setting `errors='coerce'` will convert any invalid dates to NaT (Not a Time) values.

2. \*\*pd.to\_datetime() with format parameter\*\*: If your date column has a specific format that is not automatically recognized by Pandas, you can specify the format using the `format` parameter. For example:

```python

sg\_data['date'] = pd.to\_datetime(sg\_data['date'], format='%Y-%m-%d')

```

This specifies that the date format is year-month-day.

3. \*\*Using pd.to\_datetime() with apply()\*\*: If you need more flexibility in the conversion process, you can use the `apply()` function along with a custom function to convert each element individually. For example:

```python

sg\_data['date'] = sg\_data['date'].apply(lambda x: pd.to\_datetime(x))

```

4. \*\*Using DatetimeIndex\*\*: If your DataFrame already has a datetime index, you can use the `to\_datetime()` method directly on the index. For example:

```python

sg\_data.index = pd.to\_datetime(sg\_data.index)

```

Each of these methods offers different levels of flexibility and error handling. Choose the one that best suits your specific requirements and data characteristics.

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import matplotlib.dates as mdates

import matplotlib.colors as mcolors

import matplotlib.cm as cm

# Create the figure

fig = plt.figure(figsize=(14, 8))

# Normalize stringency index for color mapping

norm = mcolors.Normalize(vmin=min(sg\_data['stringency\_index'].min(), brazil\_data['stringency\_index'].min()),

vmax=max(sg\_data['stringency\_index'].max(), brazil\_data['stringency\_index'].max()))

# Create a ScalarMappable for color mapping

scalar\_map = cm.ScalarMappable(norm=norm, cmap='copper')

# Plot for Singapore

sg\_scatter = plt.scatter(sg\_data['date'], sg\_data['total\_cases\_per\_million\_smoothed'],

c=sg\_data['stringency\_index'], cmap='copper', norm=norm, label='Singapore', alpha=0.7)

# Plot for Brazil

brazil\_scatter = plt.scatter(brazil\_data['date'], brazil\_data['total\_cases\_per\_million\_smoothed'],

c=brazil\_data['stringency\_index'], cmap='copper', norm=norm, label='Brazil', alpha=0.7)

# Adding a color bar to represent the stringency index

cbar = fig.colorbar(scalar\_map)

cbar.set\_label('Stringency Index')

# Set labels and title

plt.xlabel('Date')

plt.ylabel('Total Cases Per Million (7-Day Rolling Average)')

plt.title('COVID-19 Infection Rate with Stringency Index Over Time: Singapore vs. Brazil')

# Format the x-axis to show month and year only

plt.gca().xaxis.set\_major\_formatter(mdates.DateFormatter('%b %Y'))

plt.gca().xaxis.set\_major\_locator(mdates.MonthLocator())

plt.xticks(rotation=45)

plt.tight\_layout()

plt.show()

min\_stringency = min(sg\_data['stringency\_index'].min(), brazil\_data['stringency\_index'].min())

max\_stringency = max(sg\_data['stringency\_index'].max(), brazil\_data['stringency\_index'].max())

sg\_data['normalized\_stringency'] = (sg\_data['stringency\_index'] - min\_stringency) / (max\_stringency - min\_stringency)

brazil\_data['normalized\_stringency'] = (brazil\_data['stringency\_index'] - min\_stringency) / (max\_stringency - min\_stringency)

sg\_scatter = ax.scatter(sg\_data['date'], sg\_data['total\_cases\_per\_million\_smoothed'],

c=sg\_data['normalized\_stringency'], cmap='copper', label='Singapore', alpha=0.7)

brazil\_scatter = ax.scatter(brazil\_data['date'], brazil\_data['total\_cases\_per\_million\_smoothed'],

c=brazil\_data['normalized\_stringency'], cmap='copper', label='Brazil', alpha=0.7)

This is done to ensure that the color mapping used in the scatter plots is consistent across the two datasets.

The mcolors.Normalize() function is used to create a normalization object that maps the input data values to the range.

The code you provided:

```python

ax.xaxis.set\_major\_formatter(mdates.DateFormatter('%b %Y'))

ax.xaxis.set\_major\_locator(mdates.MonthLocator())

plt.xticks(rotation=45)

plt.tight\_layout()

plt.show()

```

is responsible for formatting the x-axis of the plot to display the dates in a specific way. Let's go through each line and explain what it's doing:

1. `ax.xaxis.set\_major\_formatter(mdates.DateFormatter('%b %Y'))`:

- This line sets the major tick formatter for the x-axis to use the `mdates.DateFormatter` class.

- The `'%b %Y'` format string tells the formatter to display the date in a format that shows the abbreviated month name (e.g., "Jan", "Feb", etc.) and the 4-digit year.

2. `ax.xaxis.set\_major\_locator(mdates.MonthLocator())`:

- This line sets the major tick locator for the x-axis to use the `mdates.MonthLocator` class.

- The `MonthLocator` class ensures that the major ticks on the x-axis are placed at the beginning of each month.

3. `plt.xticks(rotation=45)`:

- This line rotates the x-axis tick labels by 45 degrees.

- This is often done to improve the readability of the tick labels, especially when the dates are displayed in a compact format.

4. `plt.tight\_layout()`:

- This line adjusts the spacing between the plot elements (e.g., the title, labels, and tick labels) to ensure that they fit within the figure without overlapping.

5. `plt.show()`:

- This line displays the final plot.

dates = pd.concat([sg\_data['date'], brazil\_data['date']]).unique()

date\_ticks = [dates[i] for i in range(0, len(dates), len(dates) // 12)]

plt.xticks(date\_ticks, [date.strftime('%b %Y') for date in date\_ticks], rotation=45)

# Apply cutoff date filters to narrow down the analysis to the period of interest

sg\_data = sg\_data.loc[sg\_data['date'].between(cutoff\_date1, cutoff\_date)]

brazil\_data = brazil\_data.loc[brazil\_data['date'].between(cutoff\_date1, cutoff\_date)]

# Apply cutoff date filters to narrow down the analysis to the period of interest

sg\_data = sg\_data.query(f"@cutoff\_date1 <= date <= @cutoff\_date")

brazil\_data = brazil\_data.query(f"@cutoff\_date1 <= date <= @cutoff\_date")

df['dpc'] = df['new\_deaths\_smoothed'].div(df['new\_cases\_smoothed'], fill\_value=0).

By using the .loc[] indexing method, you're telling Pandas that you're modifying the original DataFrame, and not a copy. This should eliminate the SettingWithCopyWarning.

fig.legend(\*ax1.legend\_elements(), \*ax2.legend\_elements(), loc='upper left')

for country\_data, country\_name in [(sg\_data, 'Singapore'), (brazil\_data, 'Brazil')]:

# do something with country\_data and country\_name

.fillna(method='bfill'): This fills any missing values (NaNs) in the new "total\_vaccinations\_smoothed\_per\_hundred" column by using the "backfill" method, which means that the missing value is replaced with the next non-missing value.

Ffill is front fill

df['column\_name'] = df['column\_name'].interpolate(method='time')

2 closest dates take average

Use a more appropriate imputation strategy: Depending on the nature of your data and the problem you're trying to solve, the median imputation may not always be the best strategy. You could try other imputation methods, such as:

Mean imputation

Interpolation (e.g., 'time', 'index', 'linear')

Imputation based on machine learning models (e.g., KNNImputer, IterativeImputer)

The gaussian\_filter() function is from the scipy.ndimage module, and it applies a Gaussian smoothing operation to the input data.

The sigma parameter specifies the standard deviation of the Gaussian kernel, which controls the amount of smoothing. In this case, sigma=0.5 is used.

import numpy as np

import pandas as pd

def gaussian\_kernel(sigma, size):

x = np.arange(-(size // 2), size // 2 + 1, 1)

kernel = np.exp(-x\*\*2 / (2 \* sigma\*\*2))

return kernel / kernel.sum()

def gaussian\_smooth(data, sigma):

size = int(len(data) // 7) # Calculate the kernel size based on the data length

if size % 2 == 0:

size += 1 # Ensure the size is an odd integer

kernel = gaussian\_kernel(sigma, size)

smoothed = np.convolve(data, kernel, mode='same')

return smoothed

# Apply manual Gaussian smoothing to the columns

new\_cases\_smooth = gaussian\_smooth(covid\_data\_plot['new\_cases\_smoothed\_per\_million'], sigma=0.5)

total\_vaccinations\_smooth = gaussian\_smooth(covid\_data\_plot['total\_vaccinations\_per\_hundred'], sigma=0.5)

mortality\_rate\_smooth = gaussian\_smooth(covid\_data\_plot['mortality\_rate'], sigma=0.5)

# Step 1: a = [1, 2, 3, 4, 5]

# v\_flipped = [1, 0, 1]

# Result = [1 \* 1, 1 \* 0, 1 \* 1] = [1, 0, 1]

# Step 2: a = [1, 2, 3, 4, 5]

# v\_flipped = [1, 0, 1]

# Result = [2 \* 1, 2 \* 0, 2 \* 1] = [2, 0, 2]

# Step 3: a = [1, 2, 3, 4, 5]

# v\_flipped = [1, 0, 1]

# Result = [3 \* 1, 3 \* 0, 3 \* 1] = [3, 0, 3]

# Step 4: a = [1, 2, 3, 4, 5]

# v\_flipped = [1, 0, 1]

# Result = [4 \* 1, 4 \* 0, 4 \* 1] = [4, 0, 4]

# Step 5: a = [1, 2, 3, 4, 5]

# v\_flipped = [1, 0, 1]

# Result = [5 \* 1, 5 \* 0, 5 \* 1] = [5, 0, 5]

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

A black background with white text

Description automatically generated A screenshot of a computer program

Description automatically generated

import numpy as np

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

# Assuming you have the data in a pandas DataFrame

df = pd.DataFrame({

'new\_cases\_smooth': new\_cases\_smooth,

'total\_vaccinations\_smooth': total\_vaccinations\_smooth,

'mortality\_rate\_smooth': mortality\_rate\_smooth

})

# Create the heatmap

plt.figure(figsize=(12, 12))

heatmap = sns.heatmap(

df.pivot('total\_vaccinations\_smooth', 'new\_cases\_smooth', 'mortality\_rate\_smooth'),

cmap='viridis',

vmin=0,

vmax=2,

annot=False,

cbar\_kws={'label': 'Mortality Rate'}

)

# Add labels and title

heatmap.set\_xlabel('Infection Rate')

heatmap.set\_ylabel('Vaccination Rate')

heatmap.set\_title('Vaccination Rate VS Infection/Mortality Rate')

plt.show()

# Calculate the mortality rate

covid\_data['mortality\_rate'] = np.where(covid\_data['total\_cases\_per\_million'] == 0, 0, covid\_data['total\_deaths\_per\_million'] / covid\_data['total\_cases\_per\_million'])

if cases = 0 , set to 0, else set to the formula

import pandas as pd

# Assuming you have the 'covid\_data\_plot' DataFrame

original\_length = len(covid\_data\_plot)

covid\_data\_plot = covid\_data\_plot.dropna()

dropped\_rows = original\_length - len(covid\_data\_plot)

# Create a dictionary to store the dropped rows

dropped\_data = {}

for column in covid\_data\_plot.columns:

dropped\_data[column] = covid\_data\_plot.loc[covid\_data\_plot[column].isna(), column].tolist()

# Create a DataFrame from the dropped data dictionary

dropped\_df = pd.DataFrame(dropped\_data)

print(f"Number of NaN values dropped: {dropped\_rows}")

print("Dropped data:")

print(dropped\_df)

sns.heatmap(covid\_data\_plot.corr(), annot=True, cmap='YlOrRd')

annotations

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

# Assuming you have the 'covid\_data\_plot' DataFrame

# Create a correlation matrix manually

corr\_matrix = covid\_data\_plot.T.dot(covid\_data\_plot) / (len(covid\_data\_plot) - 1)

# Create a figure and axis

fig, ax = plt.subplots(figsize=(12, 12))

# Plot the correlation matrix as a heatmap

im = ax.imshow(corr\_matrix, cmap='YlOrRd', vmin=-1, vmax=1)

# Add a colorbar

cbar = fig.colorbar(im, ax=ax)

# Add labels and title

ax.set\_xticks(np.arange(len(corr\_matrix.columns)))

ax.set\_yticks(np.arange(len(corr\_matrix.columns)))

ax.set\_xticklabels(corr\_matrix.columns)

ax.set\_yticklabels(corr\_matrix.columns)

ax.set\_title('Correlation Heatmap')

# Rotate the x-axis labels for better readability

plt.setp(ax.get\_xticklabels(), rotation=45, ha="right", rotation\_mode="anchor")

# Add annotations

for i in range(len(corr\_matrix.columns)):

for j in range(len(corr\_matrix.columns)):

text = ax.text(j, i, f"{corr\_matrix.iloc[i, j]:.2f}", ha="center", va="center", color="black")

plt.show()

alternative correlation calculation using dot of AA^T

A screenshot of a computer code

Description automatically generated import pandas as pd

import numpy as np

# Log scaling function

def log\_scale\_data(data, epsilon=1e-6):

"""

Perform log scaling on the input data.

Args:

data (pandas.DataFrame): The input data to be log scaled.

epsilon (float, optional): A small value to add to the data to avoid taking the log of 0. Defaults to 1e-6.

Returns:

pandas.DataFrame: The log-scaled data.

"""

# Add a small value to the data to avoid taking the log of 0

data\_with\_epsilon = data + epsilon

# Perform log scaling

log\_scaled\_data = np.log(data\_with\_epsilon)

# Create the scaled data DataFrame

scaled\_data = pd.DataFrame(log\_scaled\_data, columns=data.columns)

return scaled\_data

# Scale the data using log scaling

covid\_data\_scaled = log\_scale\_data(covid\_data\_plot)

A screenshot of a computer program

Description automatically generated

*# Add R-squared and correlation coefficient to the plot*  
    axes[i].text(0.05, 0.95, **f**'R-squared: {r\_squared**:.2f**}\nCorrelation: {correlation\_coef**:.2f**}', transform=axes[i].transAxes, va='top')

0.05, 0.95 and trans helps to position the label R and correlation A screenshot of a computer

Description automatically generated

Here's a step-by-step explanation of what the code is doing:

coefficients = np.polyfit(filtered\_countries['cardiovasc\_death\_rate'], filtered\_countries['mortality\_rate'], 1):

This line uses the np.polyfit() function from NumPy to perform a linear regression (polynomial fit of degree 1) on the 'cardiovasc\_death\_rate' and 'mortality\_rate' columns of the filtered\_countries DataFrame.

The np.polyfit() function returns the coefficients of the best-fit polynomial, which in this case is a line (since the degree is 1).

The coefficients are stored in the coefficients variable.

polynomial = np.poly1d(coefficients):

This line creates a polynomial object polynomial from the coefficients obtained in the previous step.

The np.poly1d() function creates a 1D polynomial object that can be evaluated later.

x\_axis = np.linspace(min(filtered\_countries['cardiovasc\_death\_rate']), max(filtered\_countries['cardiovasc\_death\_rate']), 100):

This line creates a 1D NumPy array x\_axis that contains 100 evenly spaced values between the minimum and maximum values of the 'cardiovasc\_death\_rate' column in the filtered\_countries DataFrame.

This array will be used to plot the best-fit line over the range of 'cardiovasc\_death\_rate' values.

y\_axis = polynomial(x\_axis):

This line evaluates the polynomial object polynomial on the x\_axis array, and stores the resulting values in the y\_axis variable.

The y\_axis values represent the predicted 'mortality\_rate' values for the corresponding 'cardiovasc\_death\_rate' values in the x\_axis array.

# Assuming your data is in a DataFrame called 'filtered\_countries'

sns.scatterplot(x='cardiovasc\_death\_rate', y='mortality\_rate', data=filtered\_countries)

sns.regplot(x='cardiovasc\_death\_rate', y='mortality\_rate', data=filtered\_countries, color='red', line\_kws={'linewidth': 2})

more efficient using regplot to best fit

with open(os.path.join('/content', 'data', 'person\_00027.txt'), 'r') as person27:

person27\_genome = person27.read()

test(person27\_genome)

This line uses the open() function to open a file located at the path /content/person\_00027.txt.

The os.path.join() function is used to construct the file path by joining the directory /content and the filename 'person\_00027.txt'.

The 'r' mode specifies that the file should be opened for reading.

The with statement is used to ensure that the file is properly closed after the block of code is executed, even if an exception occurs.

The os.path.join() function is used to construct the file path in a platform-independent way. This function ensures that the correct directory separator (e.g., forward slash / on Unix-like systems, backslash \ on Windows) is used, regardless of the operating system.

"""

genome\_filename = f'person\_{person\_id:05d}.txt'

genome\_path = os.path.join(data\_dir, genome\_filename)

try:

with open(genome\_path, 'r') as genome\_file:

genome\_data = genome\_file.read()

return genome\_data

except FileNotFoundError:

print(f"Error: File '{genome\_filename}' not found in '{data\_dir}'.")

return None

except Exception as e:

print(f"Error reading genome data: {e}")

return None

# Using list comprehension

infected\_persons = [person\_id for person\_id in range(100) if test(get\_genome(person\_id))]

def pool(list\_of\_id):

return ''.join(get\_genome(id) for id in list\_of\_id)

def pool(list\_of\_id):

return ''.join(map(get\_genome, list\_of\_id))