

# CCT College Dublin

## Assessment Cover Page

<b>Module Title:</b>	Data Exploration & Preparation
<b>Assessment Title:</b>	CA1 Project
<b>Lecturer Name:</b>	Dr. Muhammad Iqbal
<b>Student Full Name:</b>	Daniel Bezerra Martellini
<b>Student Number:</b>	2020356
<b>Assessment Due Date:</b>	03/12/2023
<b>Date of Submission:</b>	03/12/2023

### Declaration

By submitting this assessment, I confirm that I have read the CCT policy on Academic Misconduct and understand the implications of submitting work that is not my own or does not appropriately reference material taken from a third party or other source. I declare it to be my own work and that all material from third parties has been appropriately referenced. I further confirm that this work has not previously been submitted for assessment by myself or someone else in CCT College Dublin or any other higher education institution.

---

# Data Exploration & Preparation

## CA1 Project

---

Daniel Bezerra Martellini

Student Number: 2020356

Lecturer: Muhammed Iqbal

Repository: [https://github.com/daniel-martellini-projects/CA\\_data\\_exploration](https://github.com/daniel-martellini-projects/CA_data_exploration)

<b>Introduction:</b>	<b>2</b>
<b>Question A:</b>	<b>5</b>
<b>Question B:</b>	<b>10</b>
<b>Question C:</b>	<b>11</b>
<b>Question D:</b>	<b>13</b>
<b>Question E:</b>	<b>15</b>
<b>Question F:</b>	<b>17</b>
<b>Question G:</b>	<b>18</b>
<b>Question H:</b>	<b>20</b>
<b>References:</b>	<b>21</b>

## Introduction:

In order to answer the following questions I'm going to explain how I have reached my final dataset.

My dataset is composed of 3 different datasets that I obtained on kaggle.com

- Here I'm reading the csv files for my datasets and assigning them to pandas dataframes:

```
#https://www.kaggle.com/datasets/georgesaavedra/covid19-dataset
covid_data = pd.read_csv("covid19_dataset/covid-data.csv")

#https://www.kaggle.com/datasets/gpreda/covid-world-vaccination-progress
vaccines_data = pd.read_csv("covid19_dataset/country_vaccinations.csv")

#https://www.kaggle.com/datasets/iamsouravbanerjee/world-population-dataset
world_statistics = pd.read_csv("covid19_dataset/world_population.csv")
```

The dataset I started with which I will call the first dataset, had a lot of data in it, 67 different features to be exact but many of them had too many NaN values and columns that didn't interest me, so I selected the columns that I wanted which were:

- new cases, total cases, new deaths, total deaths, country, country code, continent and date.

Then I found another dataset, which I will call the second dataset, that contained data about vaccination, also by date and country. I selected only the columns that I wanted:

- Total vaccinations, people vaccinated, and people fully vaccinated, date and country code.

Dropping features that I don't want on my final dataset:

- Here I'm reducing the size of those dataframes by only keeping the columns that I want:

```
#creating a dataframe that contains data about covid cases and deaths per date and country
columns_that_I_want = ["date", "iso_code", "location", "continent", "new_cases", "total_cases", "new_deaths", "total_deaths"]
covid_data = covid_data[columns_that_I_want]
```

```
#creating a dataframe that contains data about vaccination per date and country
columns_that_I_want = ["date", "iso_code", "total_vaccinations", "people_vaccinated", "people_fully_vaccinated"]
vaccines_data = vaccines_data[columns_that_I_want]
```

```
#creating a dataframe that contains country code and population
columns_that_I_want = ["CCA3", "2020 Population", "2022 Population"]
world_statistics = world_statistics[columns_that_I_want]
```

- Here's the size of my dataframes respectively before and after reducing the number of columns:

```
first_dataset_shape = covid_data.shape
second_dataset_shape = vaccines_data.shape
third_dataset_shape = world_statistics.shape
print(f"First dataset (covid data) is {first_dataset_shape}")
print(f"Second dataset (vaccines data) is {second_dataset_shape}")
print(f"Third dataset (population data) is {third_dataset_shape}")
```

```
First dataset (covid data) is (166326, 67)
Second dataset (vaccines data) is (86512, 15)
Third dataset (population data) is (234, 17)
```

```
first_dataset_shape = covid_data.shape
second_dataset_shape = vaccines_data.shape
third_dataset_shape = world_statistics.shape
print(f"First dataset (covid data) is {first_dataset_shape}")
print(f"Second dataset (vaccines data) is {second_dataset_shape}")
print(f"Third dataset (population data) is {third_dataset_shape}")
```

```
First dataset (covid data) is (166326, 8)
Second dataset (vaccines data) is (86512, 5)
Third dataset (population data) is (234, 2)
```

I merged the first and the second dataframes by date and country code using a "left" merge, with the first dataset being the one on the left, because I wanted to keep the data about the covid cases and deaths before the vaccination started as well, and if I used an "inner" merge I would be left with only the data from when the vaccination started.

- Merging my first dataframe and the size of it after merged:

```
#merging my dataframes based on their country code and date
first_merge_df = pd.merge(covid_data, vaccines_data, on=['iso_code', "date"], how='left')
first_merge_df.shape

(166326, 11)
```

After merging those datasets I realized that in my country and country code field I had some values that I didn't want, such as data for continents instead of countries and also for the whole world inside of the country feature. I want to clear that so I won't affect my results, and if I want to get this data back eventually, the sum of the whole continent or world, I can always regroup the data based on the continent feature in my dataset.

So to remove those unwanted rows I found a dataset with country name, country code and their total population which is a feature that I wanted anyway to be able to calculate some statistics based on their population.

Because I knew that this new dataset had all the countries that I wanted and didn't have the ones that I wanted to remove I conducted another merge but this time I used "inner" merge so I was left with only the countries present on both datasets. So I added the column that I wanted on my dataset and at the same time I filtered some results by removing undesired rows.

- Merging my new dataframe created above with my third dataset that contains data about the population around the world, here we are also removing some of the undesired countries from our previous merge, now we have 2 new columns and 12,328 rows less:

```
second_merge_df = pd.merge(first_merge_df, world_statistics, left_on=['iso_code'], right_on=['CCA3'], how='inner')
columns_to_drop = ['CCA3']
# Dropping column that would become a duplicate
second_merge_df = second_merge_df.drop(columns=columns_to_drop)
second_merge_df.shape

(153998, 13)
```

- Without using inner merge:

```
#When I use inner merge I can get rid of all of those "Countries"
#OWID stands for Our World In Data and it's one of our sources for the dataset
result = second_merge_df[second_merge_df['2020 Population'].isna()][ 'iso_code'].unique()
result

['OWID_AFR' 'OWID_ASI' 'BES' 'OWID_EUR' 'OWID_EUN' 'OWID_HIC' 'OWID_INT'
 'OWID_KOS' 'OWID_LIC' 'OWID_LMC' 'OWID_NAM' 'OWID_CYN' 'OWID_OCE' 'PCN'
 'SHN' 'OWID_SAM' 'OWID_UMC' 'OWID_WRL']
```

- Using inner merge:

```
#When I use inner merge I can get rid of all of those "Countries"  
#OWID stands for Our World In Data and it's one of our sources for the dataset  
result = second_merge_df[second_merge_df['2020 Population'].isna()]['iso_code'].unique()  
result  
  
array([], dtype=object)
```

## Question A:

a) Identify which variables are categorical, discrete and continuous in the chosen data set and show using some visualization or plot. Explore whether there are missing values for any of the variables.

I have 13 features in my dataset:

My categorical variables are the following, they are categorical because there is no way of ranking them. In this case they are all related to Countries:

- iso\_code: A 3 digit code that represents a single country and it follows the iso standardization.
- location: Name of the country in English.
- continent: Continent where the country is located.

My discrete variables are the following, they are all numeric and represent additions or counts, and can be orderer from biggest to smallest:

- new\_cases: New cases per day per country.
- total\_cases: Sum of all cases already registered per country up to that date per country.
- new\_deaths: New deaths per day per country.
- total\_deaths: Sum of all dead per country
- total\_vaccinations: Sum of the total number of vaccines applied up to that date per country.
- people\_vaccinated: Sum of the total number of people with at least one vaccine shot until that date per country.
- people\_fully\_vaccinated: Sum of the total number of people fully vaccinated up to that day per country.
- 2020 Population: Population of that respective country in the year 2020.
- 2022 Population: Population of that respective country in the year 2022.

- date: Only one entry per country per day, I can treat this variable as ordinal because I can order days from first to last.

Now I'm going to explore the dataset after my first merge:

```
first_merge_df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 166326 entries, 0 to 166325
Data columns (total 11 columns):
#   Column                Non-Null Count  Dtype
---  -
0   date                  166326 non-null object
1   iso_code              166326 non-null object
2   location              166326 non-null object
3   continent             156370 non-null object
4   new_cases             163133 non-null float64
5   total_cases           163293 non-null float64
6   new_deaths            145487 non-null float64
7   total_deaths          145451 non-null float64
8   total_vaccinations    39897 non-null  float64
9   people_vaccinated     37661 non-null  float64
10  people_fully_vaccinated 35123 non-null  float64
dtypes: float64(7), object(4)
memory usage: 14.0+ MB
```

I can see that I have 166326 rows, and also that from my 10 features 8 of them have null values.

Now I'm going to explore the dataset after the second merge:

```
second_merge_df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 153998 entries, 0 to 153997
Data columns (total 13 columns):
#   Column                Non-Null Count  Dtype
---  -
0   date                  153998 non-null object
1   iso_code              153998 non-null object
2   location              153998 non-null object
3   continent             153998 non-null object
4   new_cases             151207 non-null float64
5   total_cases           151374 non-null float64
6   new_deaths            134297 non-null float64
7   total_deaths          134459 non-null float64
8   total_vaccinations    80190 non-null  float64
9   people_vaccinated     79504 non-null  float64
10  people_fully_vaccinated 69978 non-null  float64
11  2020 Population        153998 non-null int64
12  2022 Population        153998 non-null int64
dtypes: float64(7), int64(2), object(4)
memory usage: 15.3+ MB
```

We have dropped around 12 thousand rows by using an “inner” merge, now we have 12 features and only 7 of them have NaN values.

Since 5 of those 7 features that have NaN values are the sum of all the cases at that point, I can do a forward fill based on the country code (so the forward fill doesn’t fill data for a different country).

```
: #selecting all columns with numeric value that are a sum of the values that came before them:
columns_to_ffill_by_country = ['total_cases', 'total_deaths', 'total_vaccinations', 'people_vaccinated', 'people_fully_vaccinated']
merged_with_groupby_ffill = second_merge_df
merged_with_groupby_ffill[columns_to_ffill_by_country] = second_merge_df.groupby('iso_code')[columns_to_ffill_by_country].ffill()

: merged_with_groupby_ffill.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 153998 entries, 0 to 153997
Data columns (total 13 columns):
#   Column                                Non-Null Count  Dtype
---  -
0   date                                  153998 non-null object
1   iso_code                             153998 non-null object
2   location                             153998 non-null object
3   continent                             153998 non-null object
4   new_cases                            151207 non-null float64
5   total_cases                          151374 non-null float64
6   new_deaths                           134297 non-null float64
7   total_deaths                         134459 non-null float64
8   total_vaccinations                   80190 non-null float64
9   people_vaccinated                    79504 non-null float64
10  people_fully_vaccinated               69978 non-null float64
11  2020 Population                      153998 non-null int64
12  2022 Population                      153998 non-null int64
dtypes: float64(7), int64(2), object(4)
memory usage: 15.3+ MB
```

I realized that the number of NaN values did not decrease, meaning that my data is already completely filled after it finds the first term that would be used on the forward fill.

What happens is that when covid started there wasn’t a vaccine, but we already have data about the number of cases and deaths attributed to it, and only after a while do we start to get data about vaccinations.

All of the columns that had data about Covid cases and deaths before the vaccines get NaN values because of the way that the datasets were merged.

The NaN values in our case are equal to 0 because they mean the same as no vaccines, no deaths or no new cases which would be the same as 0, so then I used .fillna(0) on my dataset and made sure that I got rid of those values:



```
merged_with_groupby_ffill_no_nan = merged_with_groupby_ffill.fillna(0)
merged_with_groupby_ffill_no_nan.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 153998 entries, 0 to 153997
Data columns (total 13 columns):
#   Column                               Non-Null Count  Dtype
---  -
0   date                                 153998 non-null object
1   iso_code                            153998 non-null object
2   location                            153998 non-null object
3   continent                           153998 non-null object
4   new_cases                           153998 non-null float64
5   total_cases                         153998 non-null float64
6   new_deaths                          153998 non-null float64
7   total_deaths                        153998 non-null float64
8   total_vaccinations                  153998 non-null float64
9   people_vaccinated                   153998 non-null float64
10  people_fully_vaccinated              153998 non-null float64
11  2020 Population                      153998 non-null int64
12  2022 Population                      153998 non-null int64
dtypes: float64(7), int64(2), object(4)
memory usage: 15.3+ MB
```

Now I have no more NaN values and my dataset is completely filled.

The last thing I want to do with my dataset is to assign the appropriate data types to my features, I know that none of my features require any Float or Double values because they are all filled with whole numbers, they are all related to the number of people and we never going to find 0.5 person, so I can change them to integer:

```
final_df = merged_with_groupby_ffill_no_nan
columns_to_convert = ["new_cases", "total_cases", "new_deaths",
                     "total_deaths", "total_vaccinations", "people_vaccinated",
                     "people_fully_vaccinated"]
final_df[columns_to_convert] = merged_with_groupby_ffill_no_nan[columns_to_convert].astype(int)
```

Here we can compare the datasets before and after changing the data type of 6 of my variables from float to int:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 153998 entries, 0 to 153997
Data columns (total 13 columns):
#   Column                               Non-Null Count  Dtype
---  -
0   date                                 153998 non-null object
1   iso_code                            153998 non-null object
2   location                            153998 non-null object
3   continent                           153998 non-null object
4   new_cases                           153998 non-null float64
5   total_cases                         153998 non-null float64
6   new_deaths                          153998 non-null float64
7   total_deaths                        153998 non-null float64
8   total_vaccinations                  153998 non-null float64
9   people_vaccinated                   153998 non-null float64
10  people_fully_vaccinated              153998 non-null float64
11  2020 Population                      153998 non-null int64
12  2022 Population                      153998 non-null int64
dtypes: float64(7), int64(2), object(4)
memory usage: 15.3+ MB
```

```
final_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 153998 entries, 0 to 153997
Data columns (total 13 columns):
#   Column                               Non-Null Count  Dtype
---  -
0   date                                 153998 non-null object
1   iso_code                            153998 non-null object
2   location                            153998 non-null object
3   continent                           153998 non-null object
4   new_cases                           153998 non-null int32
5   total_cases                         153998 non-null int32
6   new_deaths                          153998 non-null int32
7   total_deaths                        153998 non-null int32
8   total_vaccinations                  153998 non-null int32
9   people_vaccinated                   153998 non-null int32
10  people_fully_vaccinated              153998 non-null int32
11  2020 Population                      153998 non-null int64
12  2022 Population                      153998 non-null int64
dtypes: int32(7), int64(2), object(4)
memory usage: 11.2+ MB
```

Above I noticed that just by changing the Float values to Int we already reduced the memory usage by 4 MB. These values could be significantly bigger in a larger dataset, that is why it's always good to treat the data types.

This is a plot of my first and second dataset and their respective number of NaN values per feature:

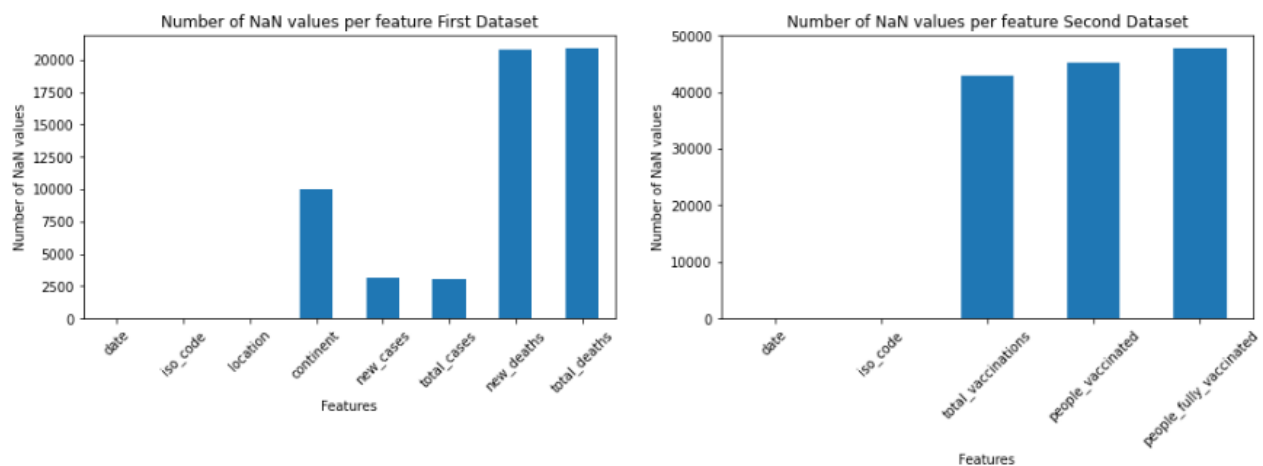
```
import matplotlib.pyplot as plt
```

```
count1 = covid_data.isna().sum()
count2 = vaccines_data.isna().sum()

fig, ax = plt.subplots(1, 2, figsize=(16, 4))
# Plotting
count1.plot(kind='bar', ax=ax[0],)
ax[0].tick_params(axis='x', rotation=45)
ax[0].set_title('Number of NaN values per feature First Dataset')
ax[0].set_xlabel('Features')
ax[0].set_ylabel('Number of NaN values')

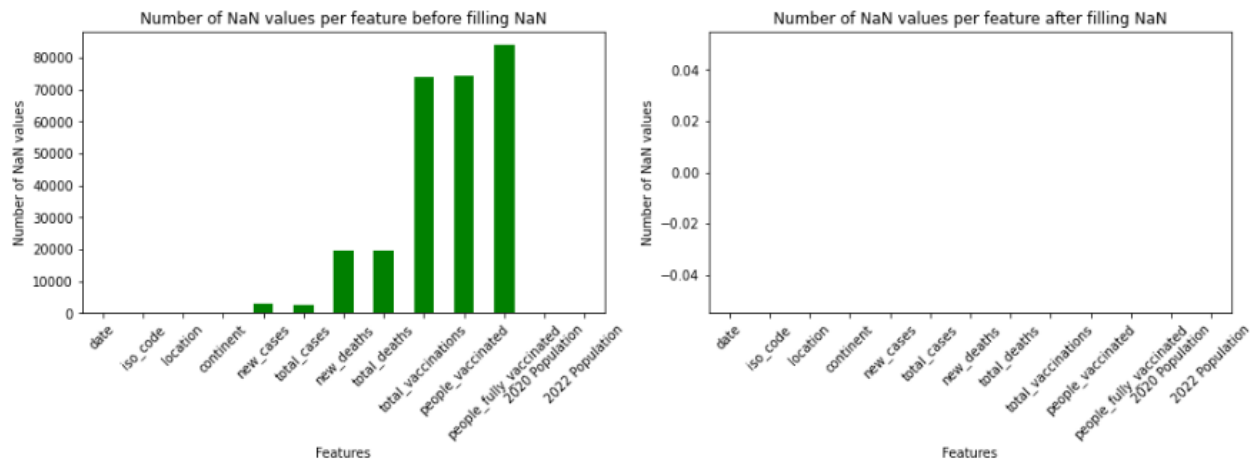
count2.plot(kind='bar', ax=ax[1],)
ax[1].tick_params(axis='x', rotation=45)
ax[1].set_title('Number of NaN values per feature Second Dataset')
ax[1].set_xlabel('Features')
ax[1].set_ylabel('Number of NaN values')

plt.show()
```



On the left we have a plot of my first and second dataset merged, and on the right we have the count of NaN values after treating our dataset, we added 0 to all empty values so it is a good thing that we didn't have any values showing there (The same code was used to plot these dataf

rames and ones above, I just change the dataframe I was plotting):



## Question B:

b) Calculate the statistical parameters (mean, median, minimum, maximum, and standard deviation) for each of the numerical variables.

I can actually get all of those parameters by using a single command in Python.

I only need to use `.describe()` on my dataframe:

```
final_df.describe()
```

	new_cases	total_cases	new_deaths	total_deaths	total_vaccinations	people_vaccinated	people_fully_vaccinated	2020 Population	2022 Population
count	1.539980e+05	1.539980e+05	153998.000000	153998.000000	1.539980e+05	1.539980e+05	1.539980e+05	1.539980e+05	1.539980e+05
mean	2.881467e+03	6.413563e+05	38.742698	13024.013156	7.282996e+06	6.626926e+06	4.873867e+06	3.821825e+07	3.885956e+07
std	1.789082e+04	3.196228e+06	182.749781	55820.639873	1.000866e+08	5.485718e+07	4.581729e+07	1.455956e+08	1.468374e+08
min	0.000000e+00	0.000000e+00	0.000000	0.000000	-2.147484e+09	0.000000e+00	0.000000e+00	5.200000e+02	5.100000e+02
25%	0.000000e+00	1.507250e+03	0.000000	21.000000	0.000000e+00	0.000000e+00	0.000000e+00	1.090156e+06	1.120849e+06
50%	5.800000e+01	1.937300e+04	0.000000	306.000000	2.555000e+03	1.587000e+03	0.000000e+00	6.979175e+06	6.948392e+06
75%	7.480000e+02	2.256570e+05	10.000000	3996.000000	1.089888e+06	7.023625e+05	3.536460e+05	2.681179e+07	2.816054e+07
max	1.368167e+06	7.926573e+07	4529.000000	958437.000000	2.142580e+09	1.269302e+09	1.234540e+09	1.424930e+09	1.425887e+09

When doing this Task I realized that I had an error in my dataset, where a specific country (China) had a negative value in the total vaccinations feature. I investigated what was happening and concluded that the column looked like it had all values correct because It was ordered in ascending order but there was a mistake with the sign, since I'm not sure the data in those rows can be trusted I decided to drop those rows.

Here I'm dropping the unwanted rows (negative values) and using the describe command again, I'm only dropping 175 rows which is not a significant number considering the total number of rows:

```
final_df = final_df[final_df['total_vaccinations'] >= 0]
final_df.describe()
```

	new_cases	total_cases	new_deaths	total_deaths	total_vaccinations	people_vaccinated	people_fully_vaccinated	2020 Population	2022 Population
count	1.538230e+05	1.538230e+05	153823.000000	153823.000000	1.538230e+05	1.538230e+05	1.538230e+05	1.538230e+05	1.538230e+05
mean	2.884643e+03	6.419710e+05	38.786657	13033.555840	9.734412e+06	5.278192e+06	3.586519e+06	3.664063e+07	3.728158e+07
std	1.790075e+04	3.197994e+06	182.849051	55851.666436	6.885100e+07	3.749117e+07	2.521022e+07	1.379564e+08	1.392643e+08
min	0.000000e+00	0.000000e+00	0.000000	0.000000	0.000000e+00	0.000000e+00	0.000000e+00	5.200000e+02	5.100000e+02
25%	0.000000e+00	1.504000e+03	0.000000	21.000000	0.000000e+00	0.000000e+00	0.000000e+00	1.090156e+06	1.120849e+06
50%	5.700000e+01	1.923400e+04	0.000000	304.000000	2.629000e+03	1.502000e+03	0.000000e+00	6.979175e+06	6.948392e+06
75%	7.505000e+02	2.261575e+05	10.000000	3957.000000	1.091909e+06	6.930000e+05	3.483080e+05	2.681179e+07	2.816054e+07
max	1.368167e+06	7.926573e+07	4529.000000	958437.000000	2.142580e+09	1.095000e+09	9.697200e+08	1.424930e+09	1.425887e+09

- count: This feature is the count of rows for each feature, they are all the same in my case because they all have values for each feature.
- mean: Sum of every row divided by the number of rows.
- std: Standard Deviation for each feature.
- min: Smallest values per feature.
- max: Biggest value per feature.
- 50%: Which is the same as our Median for each feature.
- We also have the 25th percentile and 75th percentile present per feature on the describe() method.

## Question C:

c) Apply Min-Max Normalization, Z-score Standardization and Robust scalar on the numerical data variables.

First I dropped my categorical features and then used sklearn library to apply the standardizations and scaling:

```
final_df_numeric_only = final_df_one_hot.drop(["date", "iso_code", "location", "continent"], axis=1)

#https://www.turing.com/kb/data-normalization-with-python-scikit-learn-tips-tricks-for-data-science

#Min-Max normalization
#https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html
min_max_normalization = MinMaxScaler()
df_min_max = pd.DataFrame(min_max_normalization.fit_transform(final_df_numeric_only), columns=final_df_numeric_only.columns)

#z-score standardization
#https://scikit-learn.org/stable/modules/preprocessing#standardization-or-mean-removal-and-variance-scaling
z_score_standardization = StandardScaler()
df_z_score = pd.DataFrame(z_score_standardization.fit_transform(final_df_numeric_only), columns=final_df_numeric_only.columns)

#Robust scaler
#https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.robust_scale.html
robust_scaler = RobustScaler()
df_robust_scaler = pd.DataFrame(robust_scaler.fit_transform(final_df_numeric_only), columns=final_df_numeric_only.columns)

#https://www.turing.com/kb/data-normalization-with-python-scikit-learn-tips-tricks-for-data-science
```

Here's the how the data looks after each different process:

In our first normalization, MinMax, we can easily see that our library worked, all the features have a min value of 0 and max of 1:

```
df_min_max.describe()
```

	new_cases	total_cases	new_deaths	total_deaths	total_vaccinations	people_vaccinated	people_fully_vaccinated	2020 Population	2022 Population
count	153823.000000	153823.000000	153823.000000	153823.000000	153823.000000	153823.000000	153823.000000	153823.000000	153823.000000
mean	0.002108	0.008099	0.008564	0.013599	0.004543	0.004820	0.003699	0.025714	0.026146
std	0.013084	0.040345	0.040373	0.058274	0.032135	0.034239	0.025997	0.096816	0.097669
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.000000	0.000019	0.000000	0.000022	0.000000	0.000000	0.000000	0.000765	0.000786
50%	0.000042	0.000243	0.000000	0.000317	0.000001	0.000001	0.000000	0.004898	0.004873
75%	0.000549	0.002853	0.002208	0.004129	0.000510	0.000633	0.000359	0.018816	0.019749
max	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000

In this Z-score standardization the standard deviation in every feature was brought to 1 and the mean was brought to close to 0, at first I got confused because I thought my values of the means for the features were too far from 0 but then I noticed that it was written in scientific notation and every value is indeed very close to 0 for the mean.

```
df_z_score.describe()
```

	new_cases	total_cases	new_deaths	total_deaths	total_vaccinations	people_vaccinated	people_fully_vaccinated	2020 Population	2022 Population
count	1.538230e+05	1.538230e+05	1.538230e+05	1.538230e+05	1.538230e+05	1.538230e+05	1.538230e+05	1.538230e+05	1.538230e+05
mean	1.552059e-17	1.478151e-18	4.434454e-17	-1.921597e-17	-1.182521e-17	2.069412e-17	3.695378e-18	1.034706e-17	-1.182521e-17
std	1.000003e+00	1.000003e+00	1.000003e+00	1.000003e+00	1.000003e+00	1.000003e+00	1.000003e+00	1.000003e+00	1.000003e+00
min	-1.611470e-01	-2.007424e-01	-2.121246e-01	-2.333610e-01	-1.413842e-01	-1.407854e-01	-1.422649e-01	-2.655927e-01	-2.677009e-01
25%	-1.611470e-01	-2.002722e-01	-2.121246e-01	-2.329850e-01	-1.413842e-01	-1.407854e-01	-1.422649e-01	-2.576943e-01	-2.596562e-01
50%	-1.579628e-01	-1.947280e-01	-2.121246e-01	-2.279179e-01	-1.413460e-01	-1.407453e-01	-1.422649e-01	-2.150066e-01	-2.178109e-01
75%	-1.192213e-01	-1.300237e-01	-1.574345e-01	-1.625123e-01	-1.255251e-01	-1.223010e-01	-1.284487e-01	-7.124618e-02	-6.549463e-02
max	7.626982e+01	2.458542e+01	2.455702e+01	1.692710e+01	3.097780e+01	2.906619e+01	3.832321e+01	1.006328e+01	9.971040e+00

In Robust Scaler instead of using mean and standard deviation we use median and the percentiles, robust scaler can be very useful when handling data that has a lot of outliers such as values that are much higher or lower than others, other types of normalization such as MinMax can have problems with this types of data but robust scaler can help in these cases.

```
df_robust_scaler.describe()
```

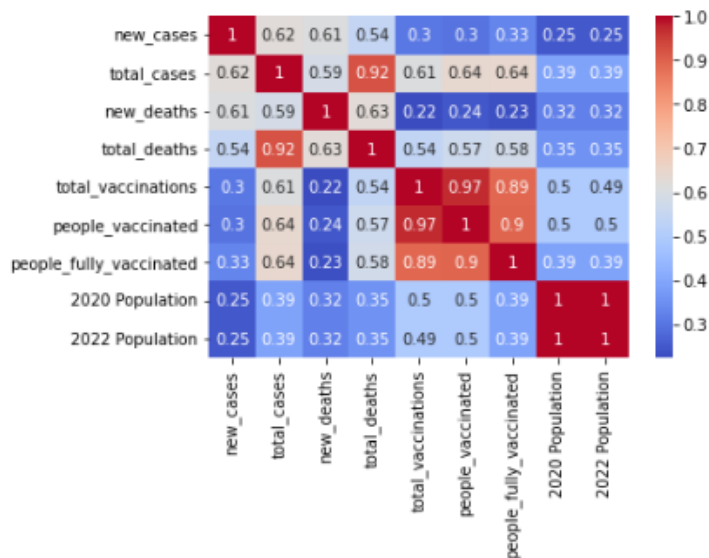
	new_cases	total_cases	new_deaths	total_deaths	total_vaccinations	people_vaccinated	people_fully_vaccinated	2020 Population	2022 Population
count	153823.000000	153823.000000	153823.000000	153823.000000	153823.000000	153823.000000	153823.000000	153823.000000	153823.000000
mean	3.767679	2.771989	3.878666	3.234135	8.912632	7.614271	10.296976	1.153171	1.121802
std	23.851763	14.235228	18.284905	14.189956	63.055618	54.099808	72.379114	5.363439	5.150367
min	-0.075949	-0.085616	0.000000	-0.077236	-0.002408	-0.002167	0.000000	-0.271315	-0.256951
25%	-0.075949	-0.078922	0.000000	-0.071900	-0.002408	-0.002167	0.000000	-0.228952	-0.215518
50%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
75%	0.924051	0.921078	1.000000	0.928100	0.997592	0.997833	1.000000	0.771048	0.784482
max	1822.931379	352.749866	452.900000	243.428100	1962.230709	1580.084413	2784.087647	55.126770	52.476148

## Question D:

d) Line, Scatter and Heatmaps can be used to show the correlation between the features of the dataset.

I started by plotting this heatmap to explore the correlation in my features:

```
import seaborn as sns
#https://medium.com/@szabo.bibor/how-to-create-a-seaborn-correlation-heatmap-in-python-834c0686b88e
heat_plot_correlational = final_df_numeric_only.corr()
sns.heatmap(heat_plot_correlational, annot=True, cmap='coolwarm')
plt.show()
```



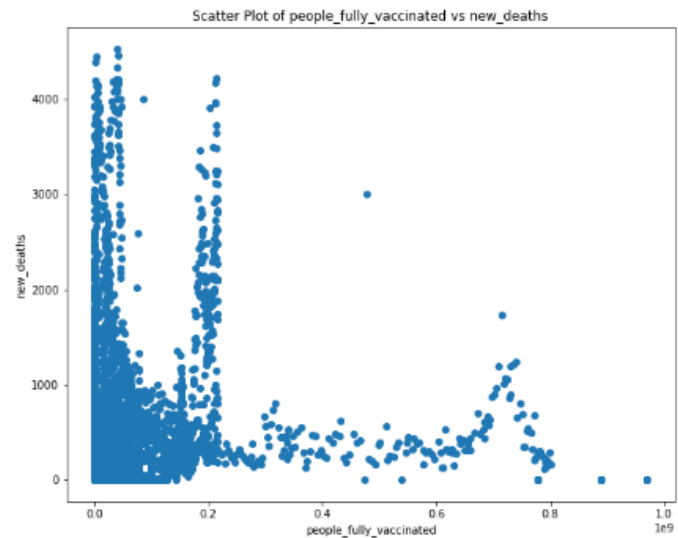
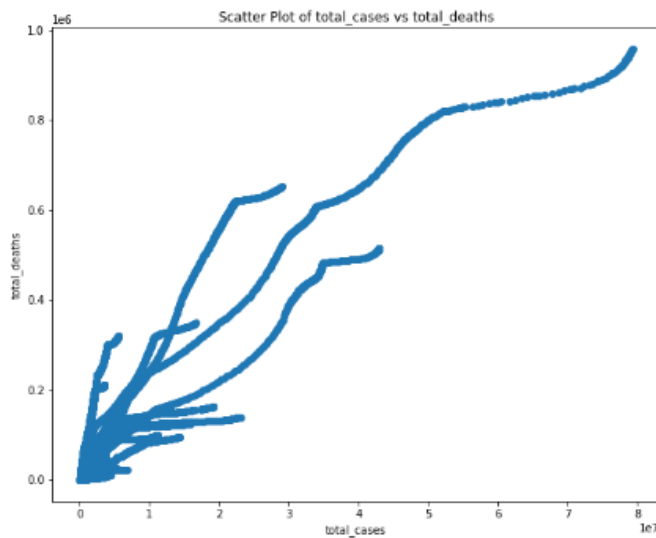
Now that I know approximately which features are correlated and which are not I can make some more plots to visualize that.

I decided to plot 2 features that are highly correlated besides other two features that are not, so we could see the different behaviors, on the left we are plotting total\_cases vs total\_deaths which is highly correlated and the second plot is people fully\_vaccinated vs number\_of\_deaths which is not highly correlated:

```
#defining features to be analyzed
columnx_1 = 'total_cases'
columny_1 = 'total_deaths'
columnx_2 = 'people_fully_vaccinated'
columny_2 = 'new_deaths'
#2 figures
fig, ax = plt.subplots(1, 2, figsize=(22, 8))
#plots
ax[0].scatter(final_df_numeric_only[columnx_1], final_df_numeric_only[columny_1])
ax[0].set_title(f'Scatter Plot of {columnx_1} vs {columny_1}')
ax[0].set_xlabel(columnx_1)
ax[0].set_ylabel(columny_1)

ax[1].scatter(final_df_numeric_only[columnx_2], final_df_numeric_only[columny_2])
ax[1].set_title(f'Scatter Plot of {columnx_2} vs {columny_2}')
ax[1].set_xlabel(columnx_2)
ax[1].set_ylabel(columny_2)

plt.show()
```



## Question E:

e) Graphics and descriptive understanding should be provided along with Data Exploratory analysis (EDA). Identify subgroups of features that can explore some interesting facts.

Based on the heatmap that we created by using a correlational matrix we can see which features affect others, and also based on the knowledge that we have about the domain.

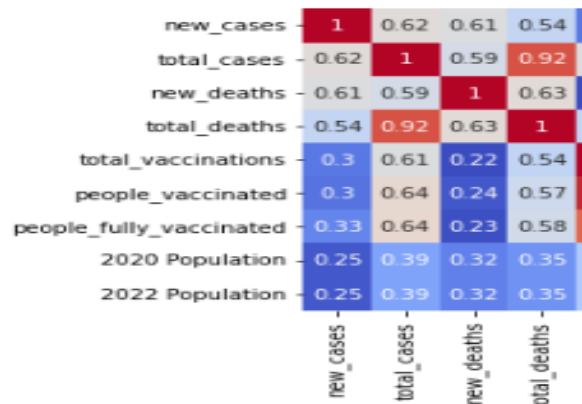
For example, I can easily affirm that the number of total\_vaccinations is going to go up whenever the number of people\_vaccinated and people\_fully\_vaccinated go up.

Here we can see that all of the vaccine features are highly correlated (red cluster on the plot):

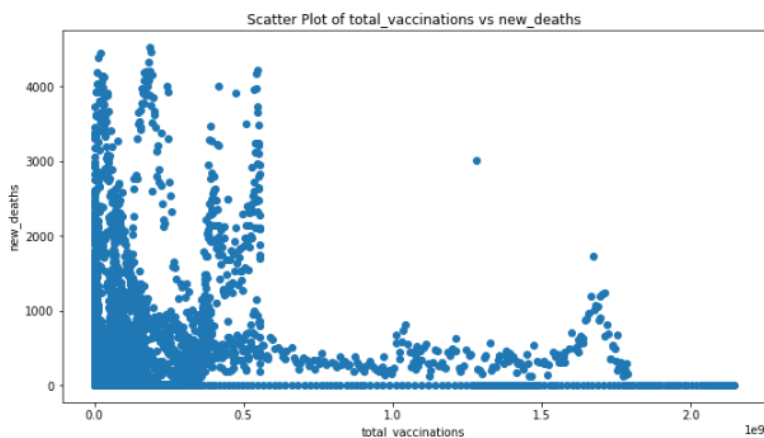
total_vaccinations	0.3	0.61	0.22	0.54	1	0.97	0.89
people_vaccinated	0.3	0.64	0.24	0.57	0.97	1	0.9
people_fully_vaccinated	0.33	0.64	0.23	0.58	0.89	0.9	1
2020 Population	0.25	0.39	0.32	0.35	0.5	0.5	0.39
2022 Population	0.25	0.39	0.32	0.35	0.49	0.5	0.39
	new_cases	total_cases	new_deaths	total_deaths	total_vaccinations	people_vaccinated	people_fully_vaccinated



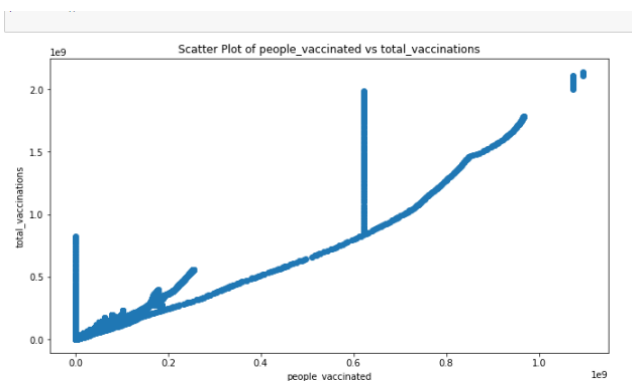
The number of deaths for example has a very high correlation (0.92) with the total number of cases:



The smallest correlation that we have in the dataset is new\_deaths vs total\_vaccinations, we can notice that the data doesn't follow any patterns:



Here's a plot of 2 highly correlated variables, total\_vaccines and people\_vaccinated:



## Question F:

f) Apply dummy encoding to categorical variables (at least one variable used from the data set) and discuss the benefits of dummy encoding to understand the categorical data.

I can easily apply dummy encoding using the pandas library, the feature that I chose was “continent”.

```
final_df["continent"].unique()
array(['Asia', 'Europe', 'Africa', 'North America', 'South America',
       'Oceania'], dtype=object)
```

I only have 6 values for continents so I believe that this is a good feature to apply one hot encoding (dummy encoding).

This is all I need to apply one hot encoding to one of my features and to create a dataframe with my old dataset and the dummies for continents :

```
: df_one_hot_regions = pd.get_dummies(final_df['continent'], prefix='continent')
final_df_one_hot = pd.concat([final_df, df_one_hot_regions], axis=1)
final_df_one_hot.shape
: (153823, 19)
```

Prefix is what is going to be written in front of every new column that was created by these lines of code. Now we have 19 columns instead of 13.

This was all created with only the code above, “prefix” parameter was merged with the value of continent:

```
final_df_one_hot.sample(5)
```

inated	people_fully_vaccinated	2020 Population	2022 Population	continent_Africa	continent_Asia	continent_Europe	continent_North America	continent_Oceania	continent_South America
0	0	77700	79824	False	False	True	False	False	False
0	0	43413	41569	False	False	False	False	True	False
11857	1072	629048	627082	False	False	True	False	False	False
0	0	125244761	123951692	False	True	False	False	False	False
539645	1043548	4498604	4736139	True	False	False	False	False	False

We can see there is only one true value per row, this can be really useful when trying to represent categorical values such as gender (for example: is\_male = True) or things like smoker and non-smoker, these values can sometimes be represented by true and false or 1 and 0 instead of a nominal category.

## Question G:

g) Apply PCA with your chosen number of components. Write up a short profile of the first few components extracted based on your understanding.

I'm using Sklearn again, now to conduct a Principal Component Analysis:

```
from sklearn.decomposition import PCA

#Creating an instance of PCA and setting how many components I want
pca = PCA(n_components=3)
df_pca = pca.fit_transform(df_z_score)

#df with the Principal Components from first to last
df_pca = pd.DataFrame(data=df_pca, columns=['pc1', 'pc2', 'pc3'])

#here we will see how each variable contributes to the PCA
loads_df = pd.DataFrame(pca.components_.T, columns=['PC1_load', 'PC2_load', 'PC3_load'], index=final_df_numeric_only.columns)
```

After adding the variance of the three components I reached 88% total explained variance, which means that we could “capture” 88% of our variance in only 3 components, before we had 9 numerical features and now we can get almost 89% of the total variance in only 3 columns.

Explained variance ratio:

```
# explained variance ratio per component
print("Variance Ratio in %:")
sum_components=0
for i, ratio in enumerate(pca.explained_variance_ratio_):
    sum_components = sum_components+ratio
    print(f"Principal component {i + 1}: {ratio:.3f}")
print(f"Total variance captured: {sum_components}")
```

```
Variance Ratio in %:
Principal component 1: 0.571
Principal component 2: 0.172
Principal component 3: 0.143
Total variance captured: 0.8872591670446299
```

Or 74% with only 2 components:

```
sum_components=0
for i, ratio in enumerate(pca.explained_variance_ratio_):
    sum_components = sum_components+ratio
    print(f"Principal component {i + 1}: {ratio:.3f}")
print(f"Total variance captured: {sum_components}")
```

```
Variance Ratio in %:
Principal component 1: 0.571
Principal component 2: 0.172
Total variance captured: 0.7437756067362186
```

Here we can see how each feature can affect the principal components positively and negatively:

loads\_df

	PC1_load	PC2_load	PC3_load
new_cases	0.260976	0.460334	-0.092389
total_cases	0.381269	0.274969	0.105724
new_deaths	0.255155	0.484095	-0.257746
total_deaths	0.359480	0.314017	0.083654
total_vaccinations	0.373481	-0.281767	0.283885
people_vaccinated	0.379263	-0.266608	0.285637
people_fully_vaccinated	0.362798	-0.196555	0.368431
2020 Population	0.297620	-0.308681	-0.551890
2022 Population	0.297770	-0.308448	-0.551682

Interestingly enough I could only reach 55% total explained variance ratio when adding the principal components when I tried using the dataframe with the one hot encoding columns created in the previous question, almost 33% less variance:

```
explained_variance_ratio = pca.explained_variance_ratio_

# explained variance ratio per component
print("Explained Variance Ratio:")
sum_components=0
for i, ratio in enumerate(explained_variance_ratio):
    sum_components = sum_components+ratio
    print(f"Principal component {i + 1}: {ratio:.4f}")
print(f"Total variance: {sum_components}")
```

```
Explained Variance Ratio:
Principal component 1: 0.3455
Principal component 2: 0.1171
Principal component 3: 0.0922
Total variance: 0.5547693683861359
```

## Question H:

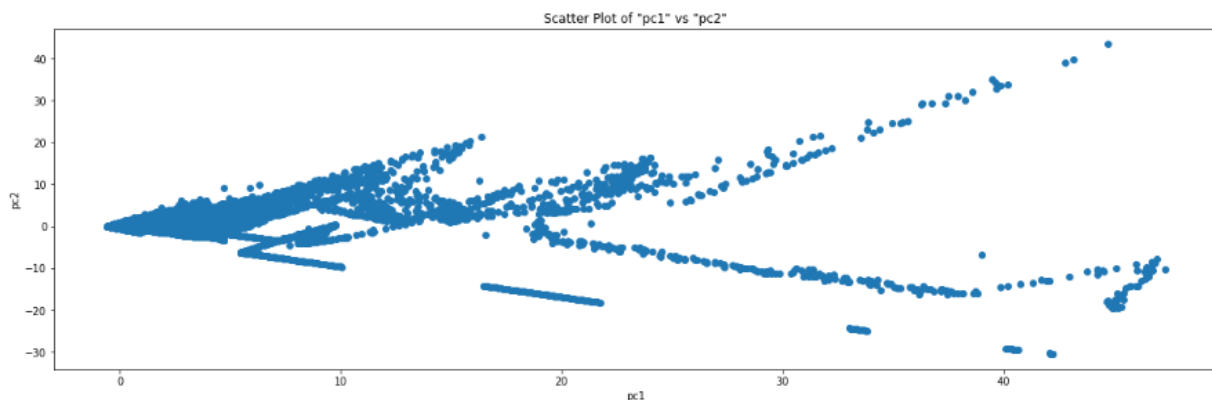
h) What is the purpose of dimensionality reduction? Explore the situations where you can gain the benefit of dimensionality reduction for data analysis.

Dimensionality reduction can have many purposes, one of them is to help visualize data in a simple manner, for example, we cannot scatter plot a dataframe with 12 features (dimensions), but if we do something to reduce the dimensionality such as conduct PCA, we can reduce the number of features into components that we can easily plot while maintaining important properties of the data:

```
#plotting my pca dataframe
fig= plt.figure(figsize=(20, 6))

plt.scatter(df_pca["pc1"], df_pca["pc2"])
plt.title(f'Scatter Plot of "pc1" vs "pc2"')
plt.xlabel("pc1")
plt.ylabel("pc2")

plt.show()
```



Dimensionality reduction also plays an important role in machine learning, where models can deal better with fewer components (features) but still need to keep most of the important data in those fewer features.

Another reason to use dimensionality reduction can be when we want or need to reduce the size (space in memory) of our dataset.

I used a dataframe with only numeric values, that were standardized, to conduct my PCA, the size of that dataframe before PCA was 10.6 MB and it had 8 columns, after PCA my new dataframe had 2 columns and only 2.3 MB, an almost 80% reduction in memory usage, while still keeping the important information:

```
#dataframe used for my PCA
df_z_score.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 153823 entries, 0 to 153822
Data columns (total 9 columns):
#   Column                Non-Null Count  Dtype
---  -
0   new_cases              153823 non-null float64
1   total_cases            153823 non-null float64
2   new_deaths             153823 non-null float64
3   total_deaths           153823 non-null float64
4   total_vaccinations     153823 non-null float64
5   people_vaccinated      153823 non-null float64
6   people_fully_vaccinated 153823 non-null float64
7   2020 Population        153823 non-null float64
8   2022 Population        153823 non-null float64
dtypes: float64(9)
memory usage: 10.6 MB
```

```
#PCA dataframe
df_pca.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 153823 entries, 0 to 153822
Data columns (total 2 columns):
#   Column  Non-Null Count  Dtype
---  -
0   pc1      153823 non-null float64
1   pc2      153823 non-null float64
dtypes: float64(2)
memory usage: 2.3 MB
```

## References:

Saavedra, G. (2022) Covid-19 dataset, Kaggle. Available at:

<https://www.kaggle.com/datasets/georgesaavedra/covid19-dataset> (Accessed: 03 December 2023).

Preda, G. (2022) Covid-19 world vaccination progress, Kaggle. Available at:

<https://www.kaggle.com/datasets/gpreda/covid-world-vaccination-progress> (Accessed: 03 December 2023).

Banerjee, S. (2022) World Population Dataset, Kaggle. Available at:

<https://www.kaggle.com/datasets/iamsouravbanerjee/world-population-dataset> (Accessed: 03 December 2023).


Pandas.dataframe.describe# (no date) pandas.DataFrame.describe - pandas 2.1.3 documentation. Available at:

<https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.describe.html> (Accessed: 03 December 2023).

Data normalization with Python Scikit-Learn: Tips for Data Science (2022) Data

Normalization with Python Scikit-Learn: Tips for Data Science. Available at:

<https://www.turing.com/kb/data-normalization-with-python-scikit-learn-tips-tricks-for-data-science> (Accessed: 03 December 2023).



*Preprocessing Data* (no date) *scikit*. Available at:

<https://scikit-learn.org/stable/modules/preprocessing#standardization-or-mean-removal-and-variance-scaling> (Accessed: 03 December 2023).

*Robust scaling: Why and how to use it to handle outliers* (no date) *Proclus Academy*.

Available at: <https://proclusacademy.com/blog/robust-scaler-outliers/> (Accessed: 03 December 2023).

Szabo, B. (2020) *How to create a Seaborn Correlation Heatmap in python?*, *Medium*.

Available at:

<https://medium.com/@szabo.bibor/how-to-create-a-seaborn-correlation-heatmap-in-python-834c0686b88e> (Accessed: 03 December 2023).

*Python pandas - get\_dummies() method* (2020) *GeeksforGeeks*. Available at:

[https://www.geeksforgeeks.org/python-pandas-get\\_dummies-method/](https://www.geeksforgeeks.org/python-pandas-get_dummies-method/) (Accessed: 03 December 2023).

*SKLEARN.DECOMPOSITION.PCA* (no date) *scikit*. Available at:

<https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html> (Accessed: 03 December 2023).

