Machine Learning Assignment-4

SECTION-A

Q1 (a)

Q1 (a) Output size after each layer:-

Convolution layer 1:-

Input size: 15 x 15 x 4

Kernel size: 5 x 5 x 4 x 1

Padding : 1

Stride: 1

Using output size = $\frac{\text{Input size} - \text{Kernel size} + 2 \times \text{Padding}}{\text{Stride}} + 1$

for both height and width

Output size = $\frac{15 - 5 + 2 \times 1}{1} + 1 = 13$

So, output size after 1st convolution layer is 13 x 13 x 1

Max pooling layer

Kernel size: 3 x 3

Stride: 2

Output size = $\frac{\text{Input size} - \text{Pooling size}}{\text{Stride}} + 1$

Output size = $\frac{13 - 3}{2} + 1 = 5 + 1 = 6$

So, the output size is 6 x 6 x 1

Convolution layer 2:

Input size: 6 x 6 x 1 (from max pooling layer)

Kernel size: 5 x 3 x 4 x 1

Padding : 2

stride : 2

PAGE

Here Kernel depth of 4 should match the number of Input channels, which is 1 from the output layer of max pooling

So, Assuming Kernel should be $5 \times 3 \times 1 \times 1$

$$\text{Output size} = \frac{\text{Input size} - \text{Kernel size} + 2 \times \text{Padding}}{\text{Stride}} + 1$$

$$\text{Output height} = \frac{6 - 5 + 2 \times 2}{2} + 1 = \frac{5}{2} + 1 = 3.5$$

Since we can't have a Half pixel, we take the floor value of height, output height = 3

$$\text{Output width} = \frac{6 - 3 + 2 \times 2}{2} + 1 = 4 \quad \text{(Floor value)}$$

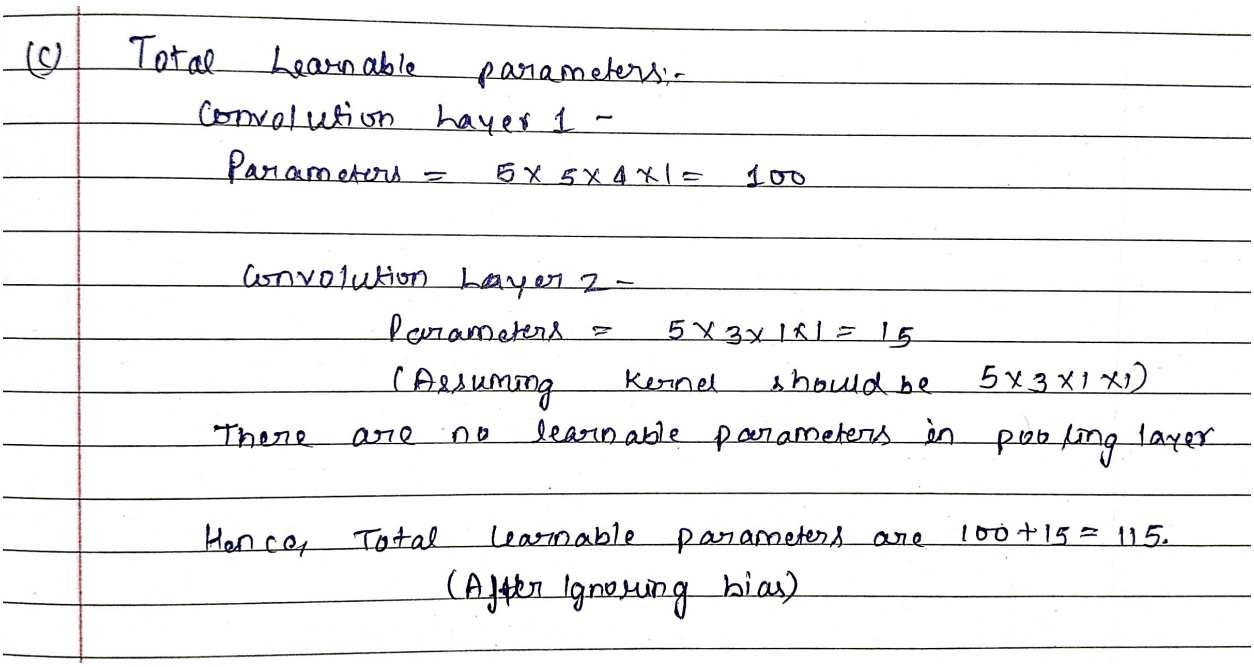
So final output size is $3 \times 4 \times 1$.

(a) (b) The main purpose of pooling is to reduce the dimensionality of the feature map, which in turn reduces the number of parameters to learn and the amount of computation to be performed.

Invariance to local translation can be a very useful property if we care more about whether some feature is present than exactly where it is.

Since pooling is used for downsampling, it can be used to handle inputs of varying sizes

(a) (c)



(b) No, It is not possible for the k-means algorithm to revisit a configuration because at each iteration it aims to minimize the sum of squared distances between data points and their assigned cluster centroids. The optimization process involves iterative updates to the centroids and reassignment of data points to clusters. The algorithm converges when there is no further improvement in the sum of squared distances. Once convergence is reached, the centroids and cluster assignments are fixed.

Basically algorithm's convergence is based only on the fact that error (sum of squared distances between data points and their assigned cluster centroids) decreases each time we calculate the new clusters which which in turn proves the fact that Kmeans will not revisit the same configuration again.

Let's assume a configuration A with error x, and next configuration B with error y (It is very obvious that x>y) now if we assume that it will go back to the same configuration then x should be less than y (x<y) again which contradicts the previous fact.

(c) KNN is a non-parametric learning algorithm that classifies a data point based on how its neighbors are classified. In contrast, a neural network is a parametric model that learns a function to map inputs to outputs.

In case of kNN there is no concept of decision boundary hence, it is not straightforward to model a kNN algorithm using a standard feedforward neural network structure because kNN's decision boundaries can be arbitrarily complex and depend on the entire dataset. kNN requires all the data to make a decision at runtime, whereas a neural network abstracts the data into parameters during training and discards the training data.

Even if we hypothetically assume that there exists a decision boundary that can be used to model neural network as kNN then we would end up having too many assumptions.

However, one might consider using a type of network which can learn to cluster inputs in a way somewhat analogous to kNN. Yet, it still does not model the kNN algorithm directly.

(d)

A linear filter is a filter that operates on a local region of the input (like an image) and captures linear relationships in the data. It applies a linear transformation which means the output is a linear combination of the input pixel values. Linear filters are useful for tasks like edge detection, blurring, and sharpening.

A non-linear filter involves a non-linear transformation of the input data. In CNNs, non-linearities are introduced after the convolutional layers through activation functions like ReLU (Rectified Linear Unit) and sigmoid. In this context pixels may not be linearly related so these non-linear functions allow CNNs to capture more complex patterns in the data, which is essential for tasks like image classification where the relationship between input pixels and the target can be highly non-linear.

SECTION-B

I have considered the sample inputs as simple numpy 2D arrays for easy interpretation.

As shown in the below image input data is 2D numpy array, below output is calculated using padding=0, size of convoluted output would be calculated as :

output_height = (input_height - kernel_size + 2*padding)/(stride) + 1

output_width = (input_width - kernel_size + 2*padding)/(stride) + 1

output_height = (3-2 + 2*0)/1 + 1 = 2

output_width = (3-2 + 2*0)/1 + 1 = 2

As I have also considered pooling size = 1, so the pooled_output would also be same.

```
Input data:
 [[1. 6. 2.]
 [5. 3. 1.]
 [7. 0. 4.]]

Kernel data:
 [[ 1.  2.]
 [-1.  0.]]
Convoluted Output:
 [[8. 7.]
 [4. 5.]]

Pooled Output:
 [[8. 7.]
 [4. 5.]]
```

If we consider pooling size as 2

the it will report only 8 as answer from whole convoluted output:

```
Pooled Output:
 [[8.]]
```

Now, as we do not have any true labels for the random generated input so we can't calculate the error in this case so I have initialized the gradient matrix (this matrix contains the gradient of loss with respect to the inputs) and another matrix contains gradient of loss with respect to the kernel:

```
d_out_conv = np.random.rand(*convoluted_output.shape)
d_out_pool = np.random.rand(*pooled_output.shape)
```

Derivative of loss function with respect to input in CNN:

for a 2x2 kernel and 3x3 input output would be:

kernel = [[k11, k12],

[k21,k22]]

Input_size = [[x11,x12,x13],

[x21,x22,x23],

[x31,x32,x33]]

y11 = k11*x11 + k12*x12 + k21*x21 + k22*x22

y12 = k11*x12 + k12*x13 + k21*x22 + k22*x23

y21 = k11*x21 + k12*x22 + k21*x31 + k22*x32

y22 = k11*x22 + k12*x23 + k23*x32 + k22*x33

gradient of Error E w.r.t x11 would be calculated as:

dE/dx11 = (dE/dy11)*(dy11/dx11) + (dE/dy12)*(dy12/dx11) + (dE/dy21)*(dy21/dx11) + (dE/dy22)*(dy22/dx11)

And that's how I updated the weight in back propagation:

```
#updating the above defined gradients
for h in range(d_out.shape[0]):
    for w in range(d_out.shape[1]):
        h_start, w_start = h * stride, w * stride
        h_end, w_end = h_start + F, w_start + F
        # Gradient with respect to input
        d_padded_input[h_start:h_end, w_start:w_end] += kernel * d_out[h, w]
        # Gradient with respect to kernel
        d_kernel += padded_input[h_start:h_end, w_start:w_end] * d_out[h, w]
```

Output:

```
Gradient wrt Input of Convolution Layer:
[[-1.18892259 -1.14555677 -0.3373417   0.84565355  0.19630572  0.        ]
 [ 0.13220805 -0.47641164 -0.50065229  1.30155467  0.35278225  0.        ]
 [ 0.3485254   1.06957738  0.42416781  1.57775112  0.38721976  0.        ]
 [-0.20877739  0.98880429  0.70382621  0.93675658  0.23339039  0.        ]
 [-0.05759732  0.20538442  0.1573543   0.1388145   0.03428254  0.        ]
 [ 0.          0.          0.          0.          0.          0.        ]]

Gradient wrt Kernel of Convolution Layer:
[[0.01270349 0.19022803 0.56392521 0.36424419 0.51750811]
 [0.16985332 0.90016178 1.37713656 0.70828442 0.91237726]
 [0.2792185  1.3328916  0.97306804 0.62134424 0.46766707]
 [0.21017175 0.93580259 0.63486386 0.99513591 0.60241993]
 [0.24044749 1.081368   0.76120356 0.8218792  0.53790473]]

Gradient wrt Input of Pooling Layer:
[[0.94187978 0.99107855]
 [0.8801611  0.36552241]]
```

SECTION-C

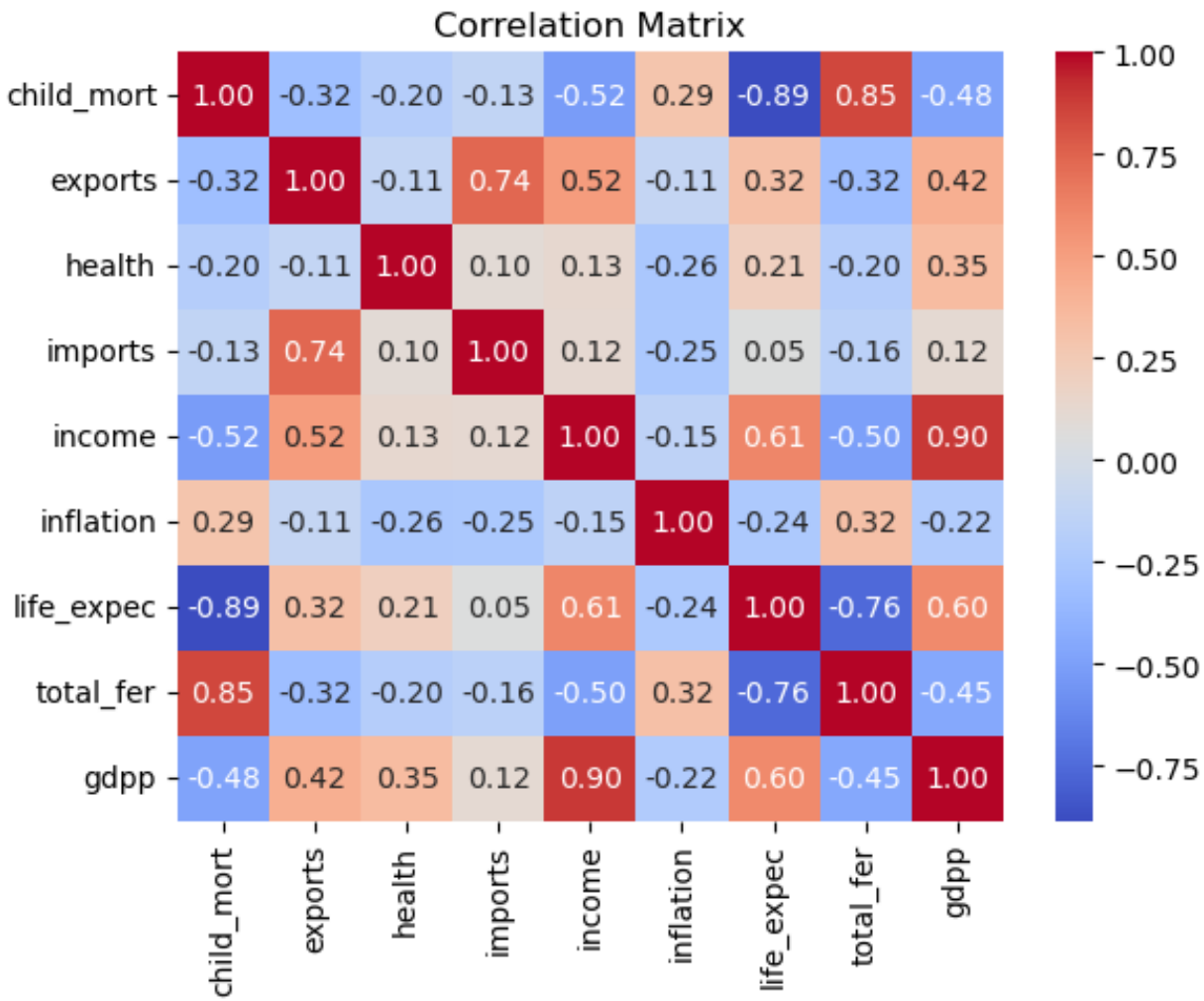
(a) **Exploratory data analysis:**

Basic information and statistics about the data:

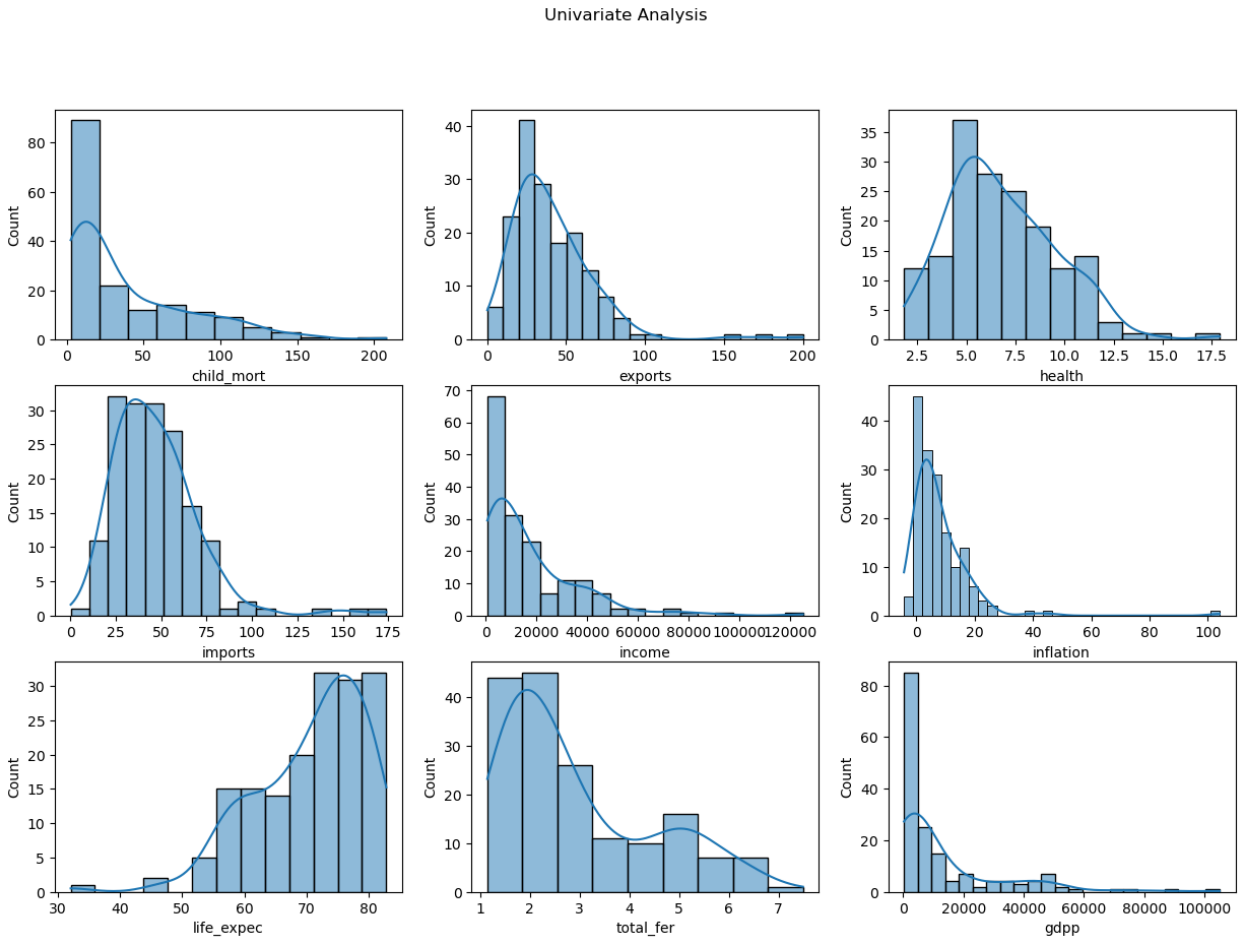
```
              country  child_mort  exports  health  imports  income  \
0         Afghanistan        90.2     10.0    7.58     44.9    1610
1             Albania        16.6     28.0    6.55     48.6    9930
2             Algeria        27.3     38.4    4.17     31.4   12900
3              Angola       119.0     62.3    2.85     42.9    5900
4  Antigua and Barbuda        10.3     45.5    6.03     58.9   19100

   inflation  life_expec  total_fer   gdpp
0       9.44        56.2       5.82    553
1       4.49        76.3       1.65   4090
2      16.10        76.5       2.89   4460
3      22.40        60.1       6.16   3530
4       1.44        76.8       2.13  12200
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 167 entries, 0 to 166
Data columns (total 10 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   country     167 non-null    object
 1   child_mort  167 non-null    float64
 2   exports     167 non-null    float64
 3   health      167 non-null    float64
 4   imports     167 non-null    float64
 5   income      167 non-null    int64
 6   inflation   167 non-null    float64
```

There were no null/missing values in the data.

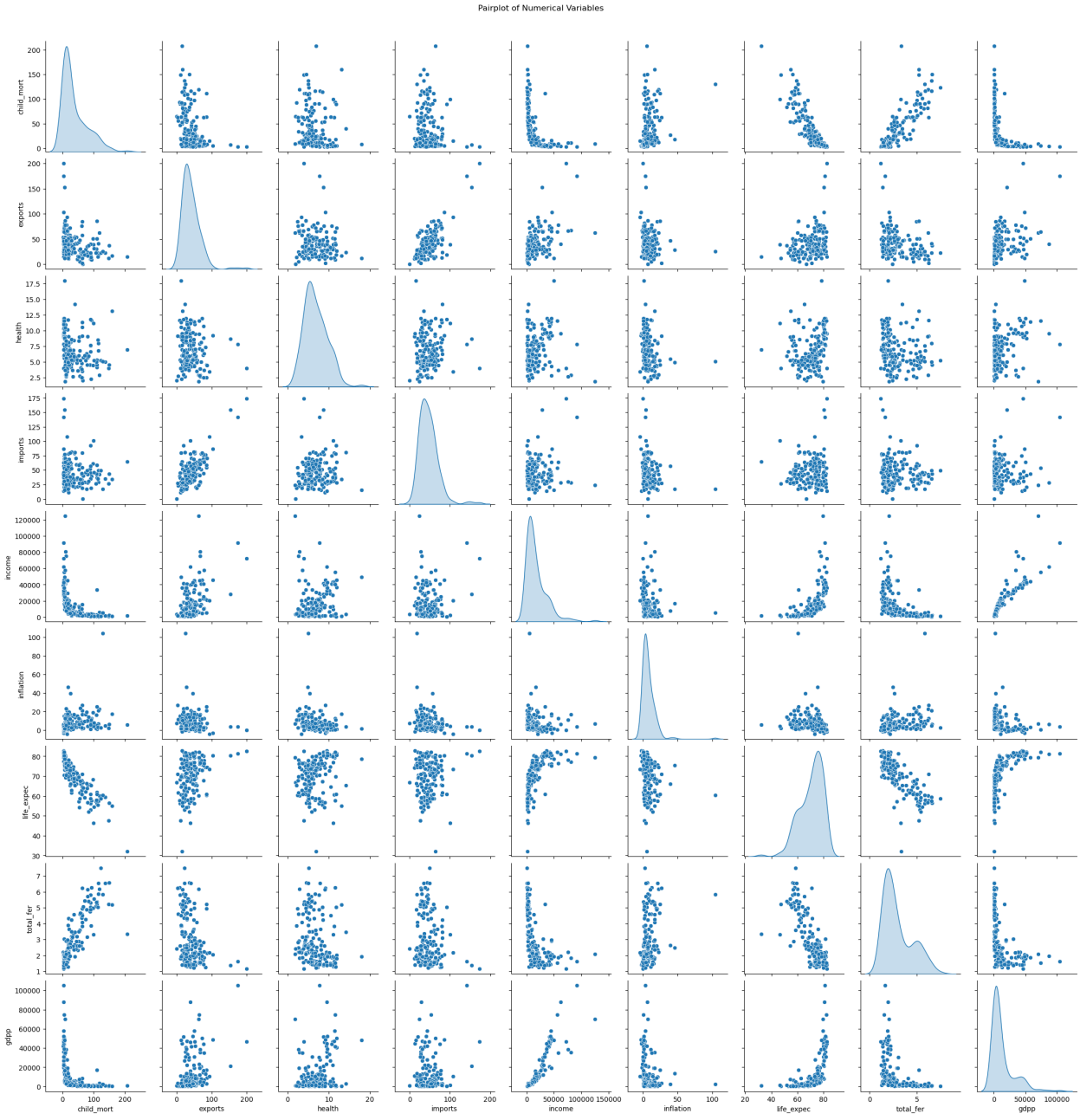
Correlation matrix:



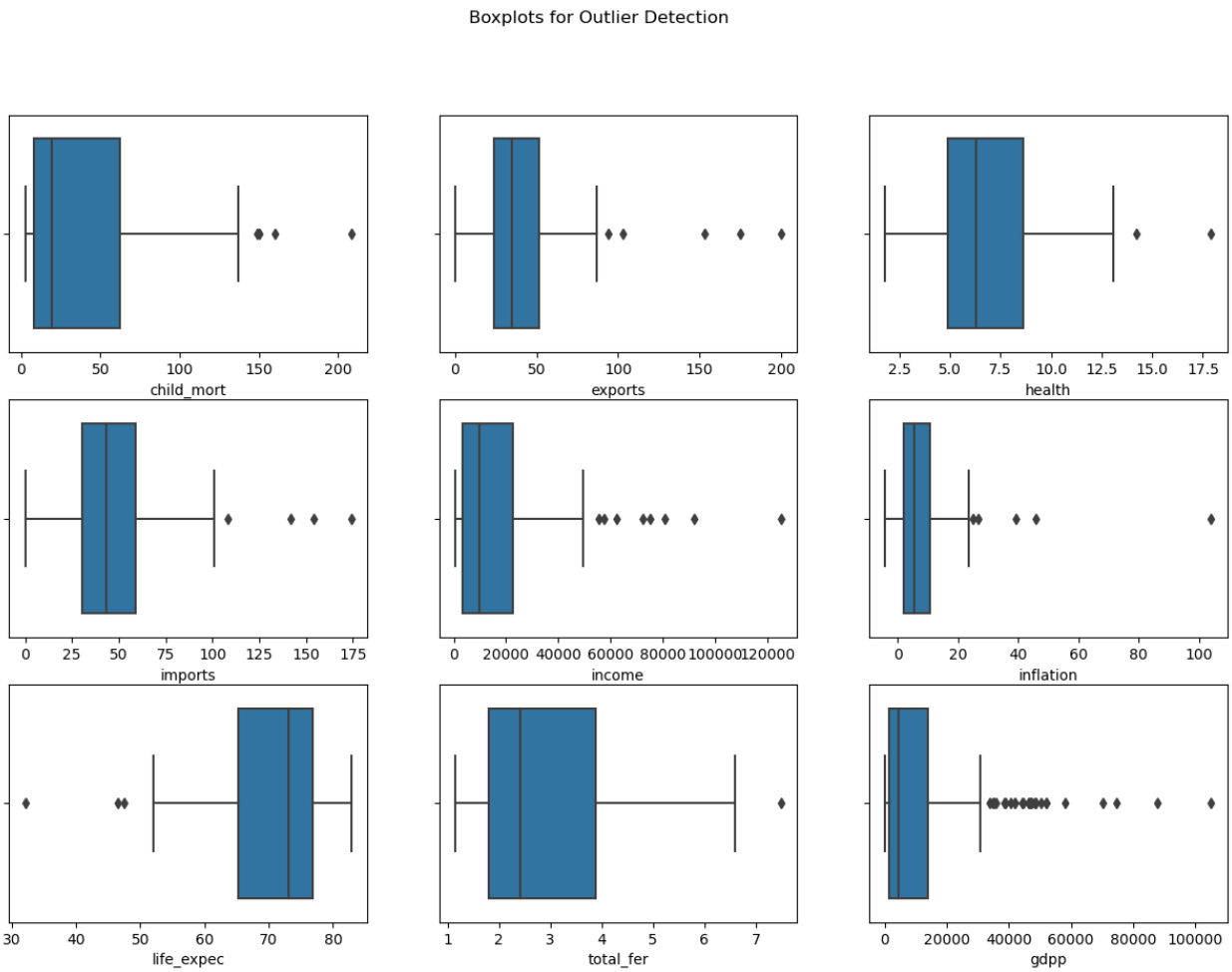
Univariate analysis:


Univariate Analysis
Count
child_mort
exports
health
imports
income
inflation
life_expec
total_fer
gdpp

Pairplots:



Boxplots for outlier detection:



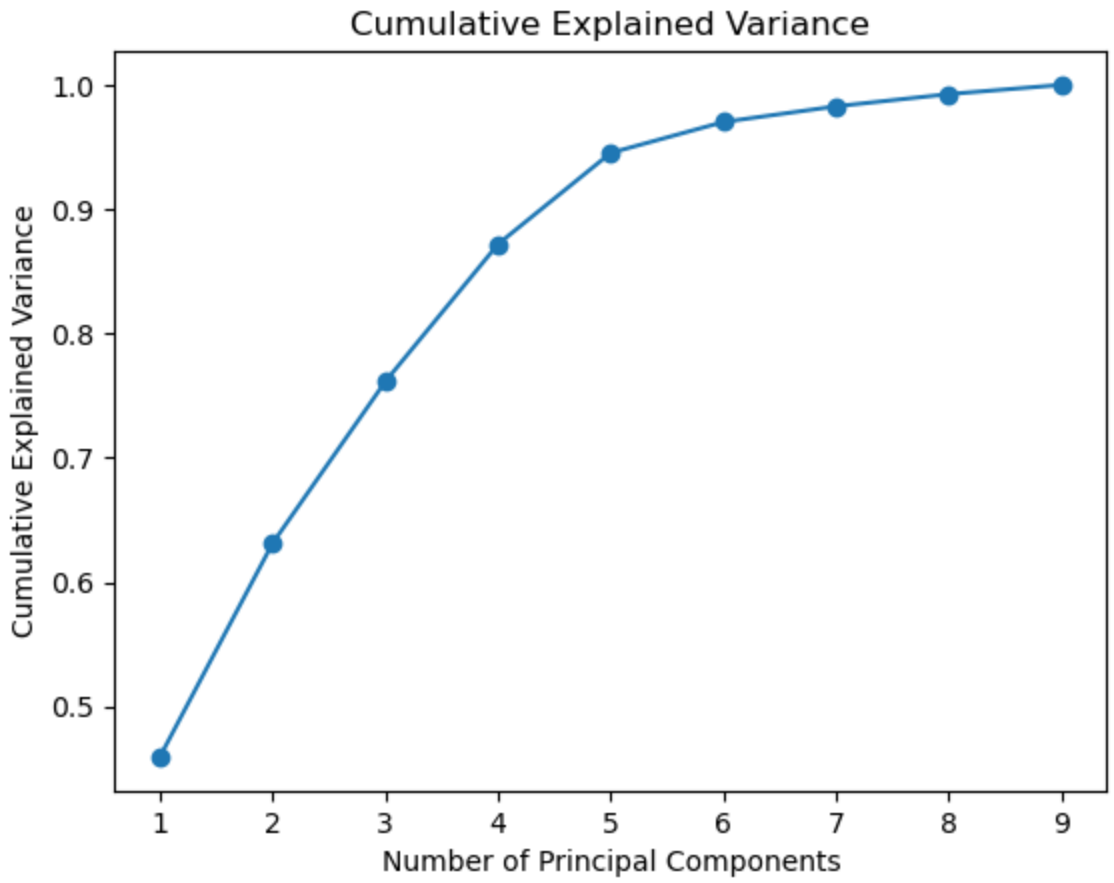
Used Standard Scaler to standardize the features.

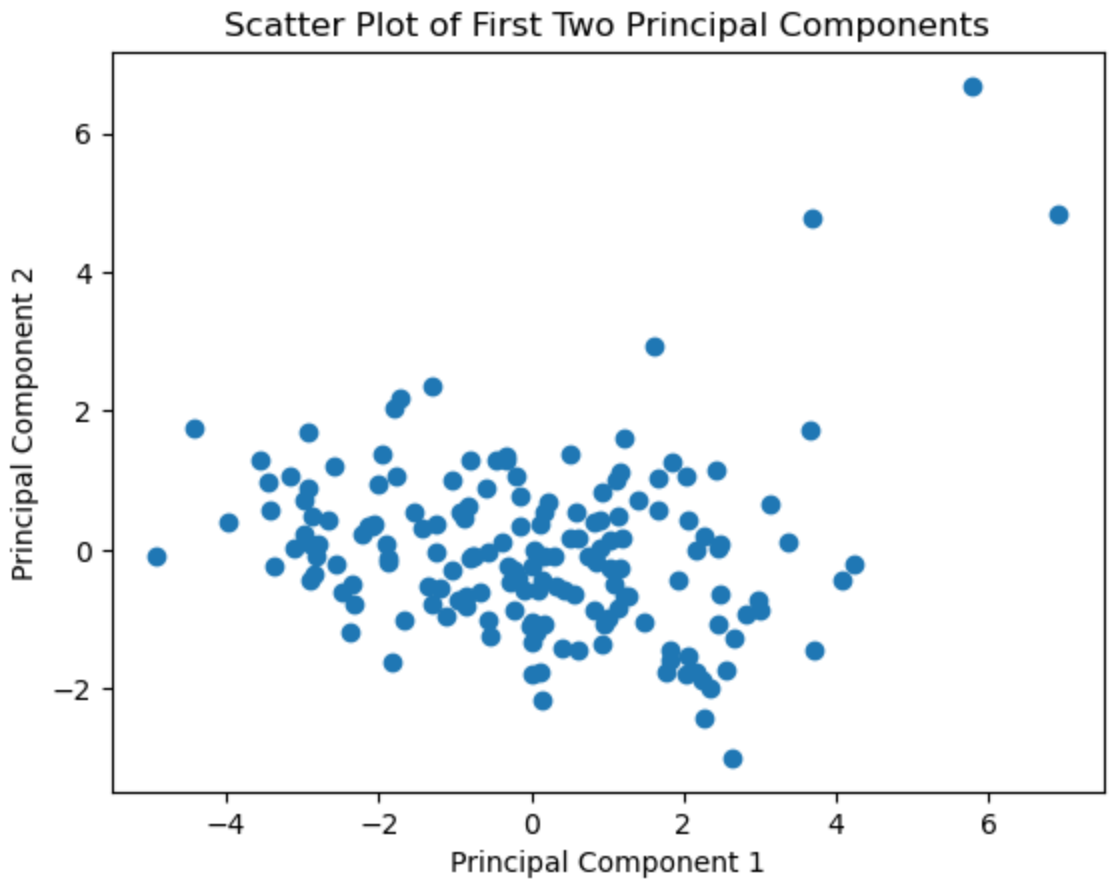
```
scaler = StandardScaler()

# Fit and transform the data
scaled_features = scaler.fit_transform(features)

# Create a new DataFrame with the standardized features
scaled_data = pd.DataFrame(scaled_features, columns=features.columns)
```

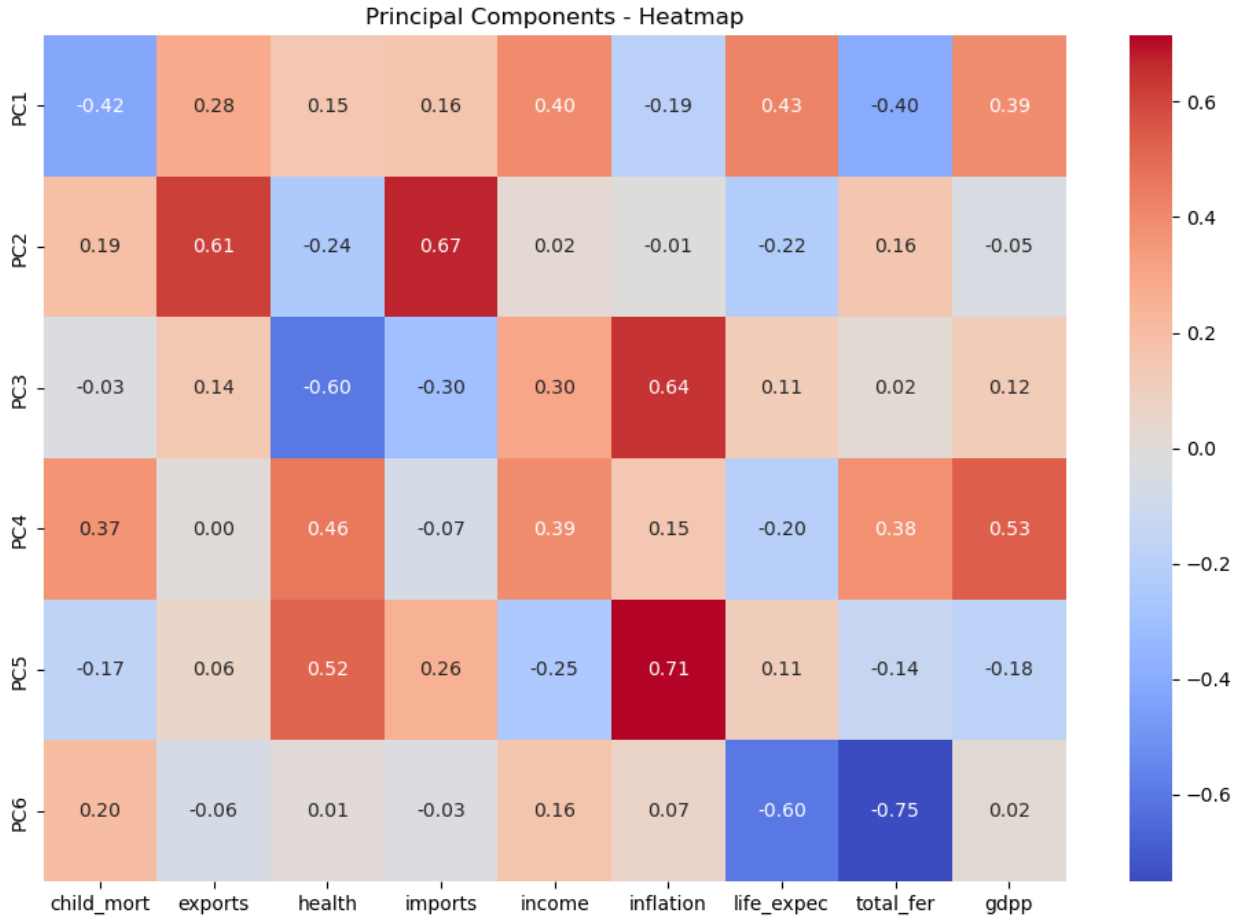
(b) Determined the optimal number of components based on a desired explained variance threshold of 0.95, got optimal number of components 6 as clear from the below plot.

(It will vary according to the threshold choosen)

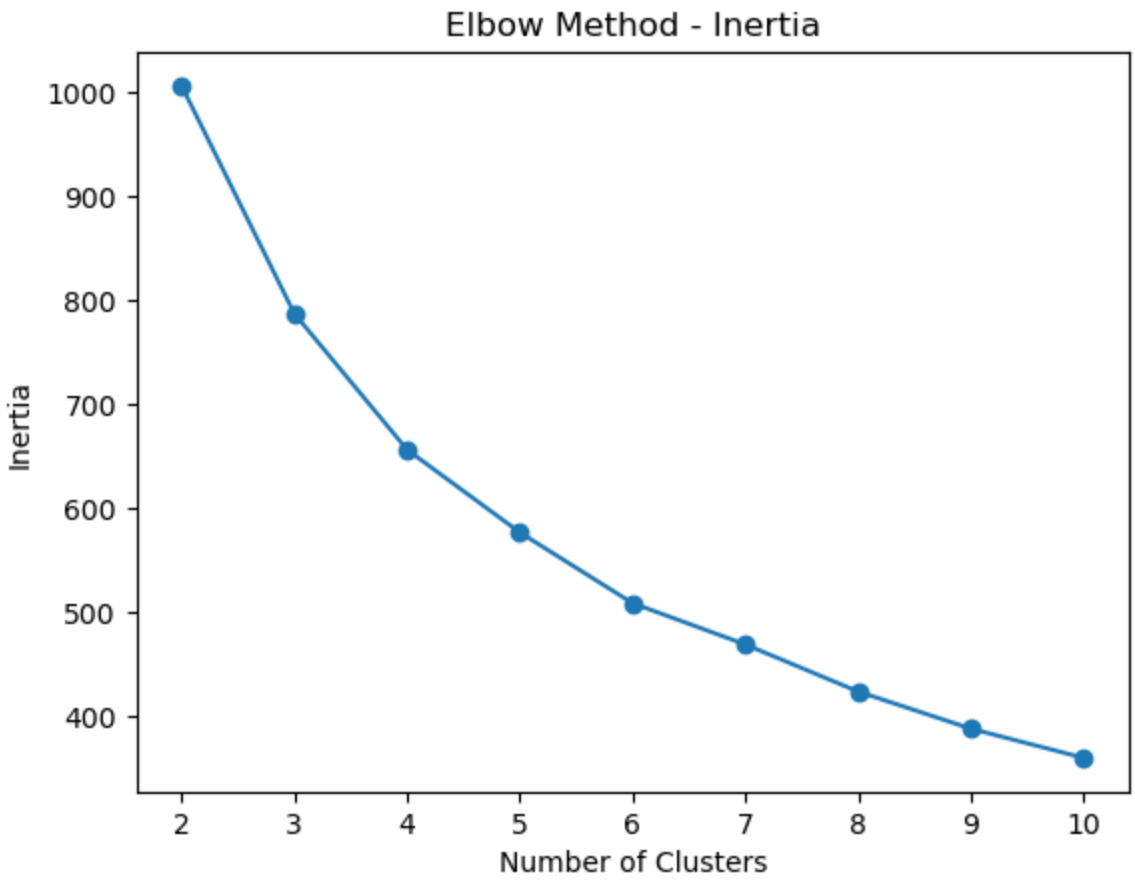
Plot for Cumulative explained variance vs number of components:

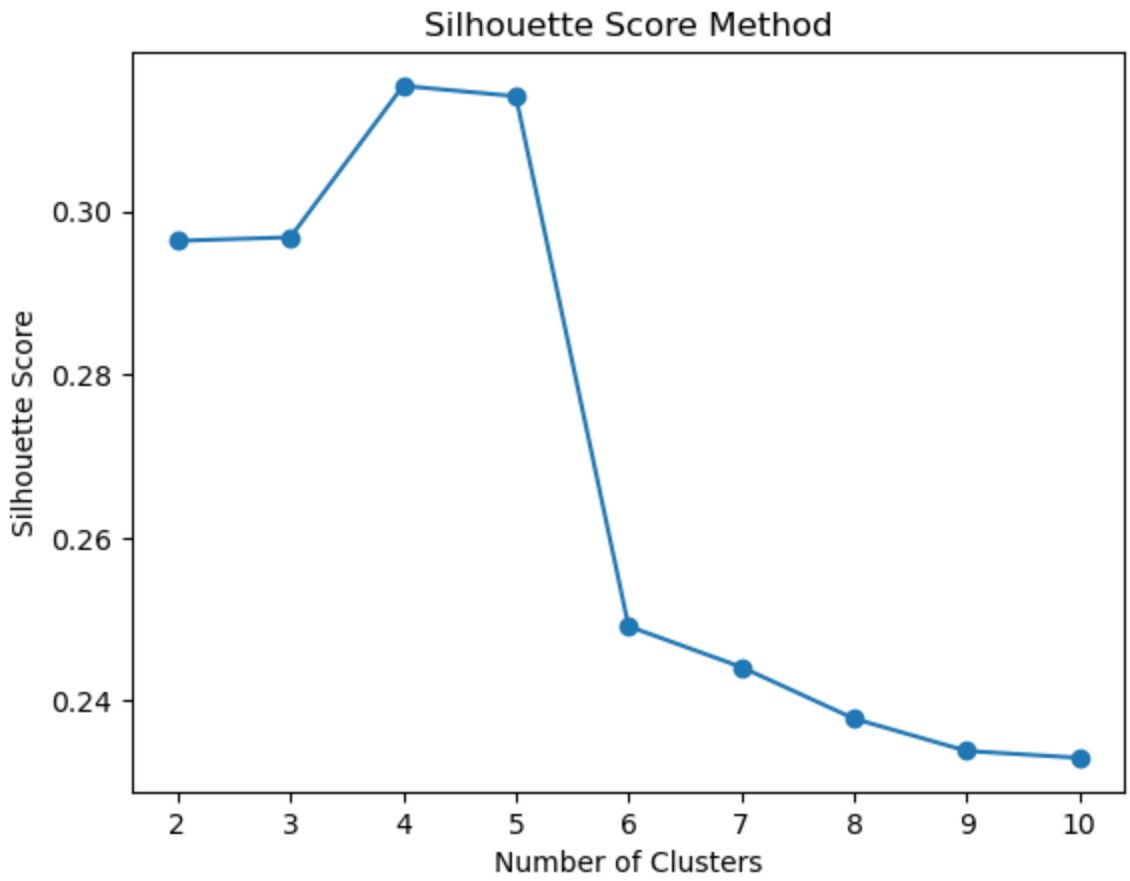



Scatter Plot of First Two Principal Components
Principal Component 2
Principal Component 1
6
4
2
0
−2
−4
−2
0
2
4
6



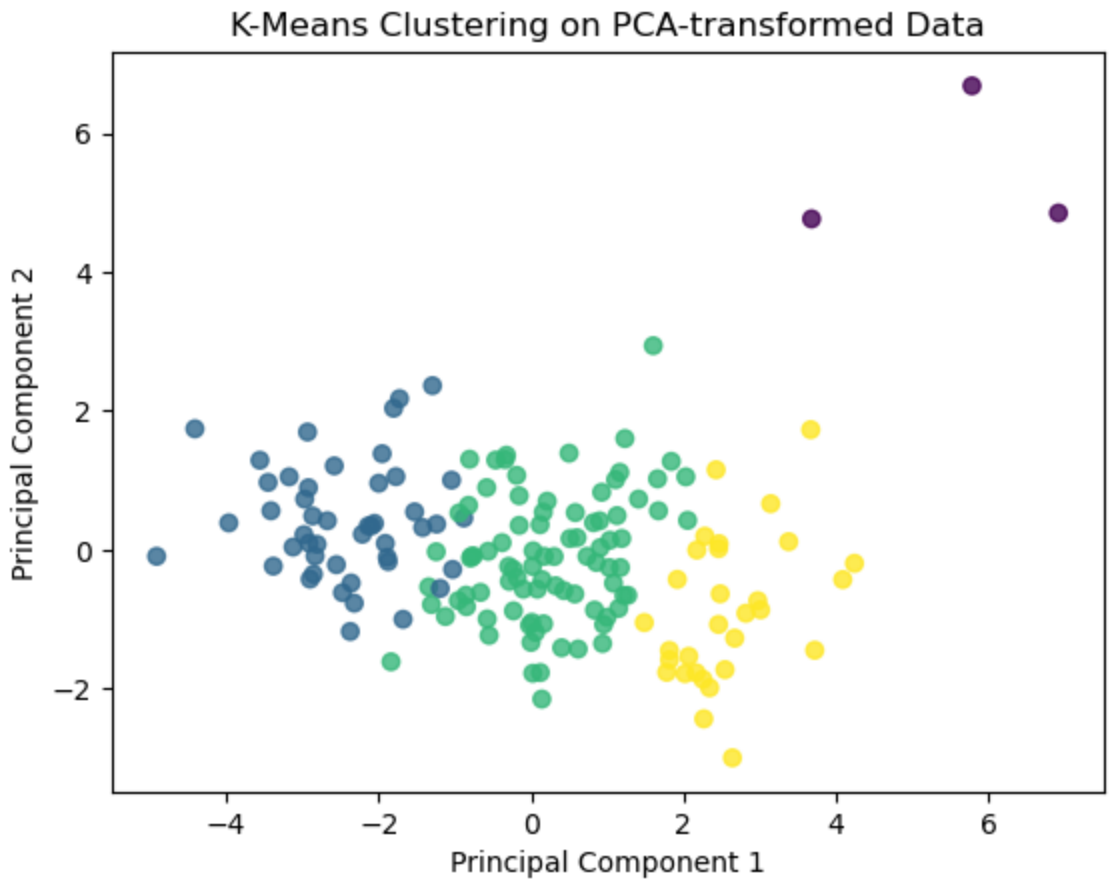
(c) Applied both elbow method and silhouette score for finding optimal number of clusters from below plots:


Elbow Method - Inertia
Inertia
1000
900
800
700
600
500
400
2
3
4
5
6
7
8
9
10
Number of Clusters



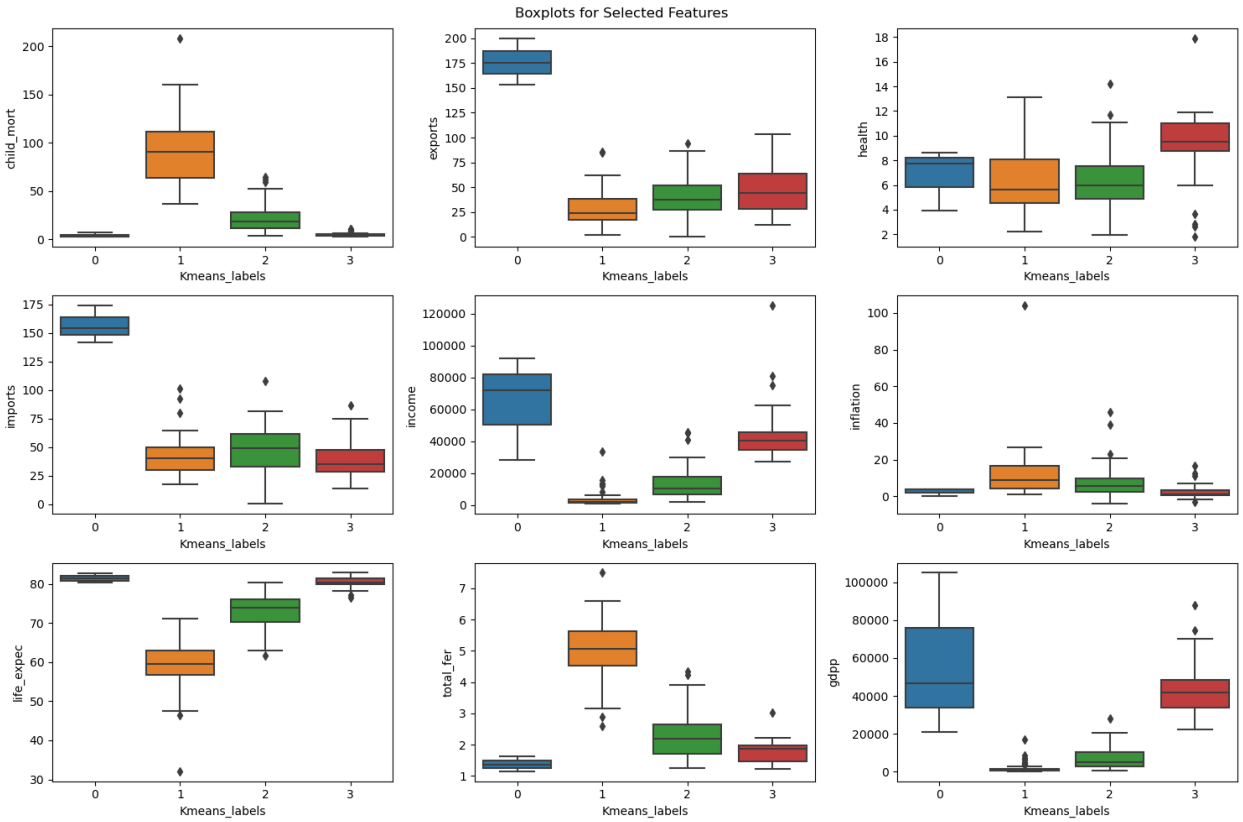
As we can observe from the plots there is clear elbow at 3 and 4 clusters and maximum silhouette score at 4 clusters.

Hence I have taken final optimal number of clusters to be 4 only.



Added the generated Kmeans label for each data point in original dataset.

Boxplot for each feature vs Kmeans label:



Observation:

Cluster 0 has a significantly higher median and wider spread in the export feature compared to other clusters, suggesting more variability and generally higher values for this feature within this cluster. Clusters 1, 2, and 3 have similar medians, but cluster 2 has some outliers.

i) Cluster 1,2 have low GDP and lower life_expectancy

ii) Cluster 1,2 have low income and higher child_mort.

iii) Cluster 0,3 have high income and lower child_mort and life_expectancy

iv) Cluster 0,3 have high GDP, and better health.