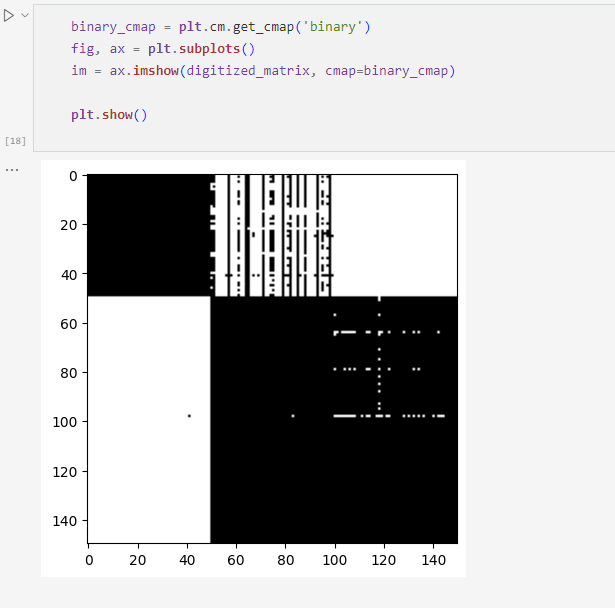
# Task 1:

In Task 1 we aim to analyze the correlation between variables in each dataset. The code first loads the dataset from a text file and stores it in a NumPy array using the **np.loadtxt()** function. The **np.corrcoef()** function is then used to calculate the correlation coefficient matrix of the data. The means of the correlation matrix rows are then calculated using the **np.mean()** function. The correlation matrix is then mean-subtracted by reshaping the means vector using **reshape(-1,1)** and subtracting it from the correlation matrix. This mean-subtracted matrix is then digitized by thresholding at zero using **(matrix\_mean\_subtracted > 0).astype(int)**. The **plt.cm.get\_cmap()** function is then used to obtain a binary colormap, which is used to visualize the digitized matrix as a black and white image using **imshow()**.

Next, the correlation matrix is normalized by dividing each row with its maximum value using **np.max(matrix, axis=1)** and stored in **scaled\_matrix**. The values of the scaled matrix are then multiplied by 255 to scale them between 0 and 255, which is the range for RGB values. The resulting matrix is then converted to a 3D array of unsigned integers of size **matrix.shape[0] x matrix.shape[1] x 3** using **np.zeros()** and **dtype=np.uint8**. The green channel of the 3D array is then set to the scaled matrix values using **rgb[:, :, 1] = scaled\_matrix.astype(np.uint8)**. Finally, the RGB image of the correlation matrix is displayed using **imshow()**.

Overall, the code loads the dataset, calculates the correlation coefficient matrix, visualizes the decentralized matrix using a binary colormap, and displays the RGB image of the correlation matrix, providing a useful way to analyze the correlation between variables in a given dataset.

Graphical user interface, application

Description automatically generated

# Task 2:

Task 2 involves permuting the Data Matrix, displaying a color-coded image of the permuted Data Matrix, recovering the image clusters using the Signature technique, rearranging the Similarity Matrix by the signature value of each row, applying Task 1 on the rearranged matrix, and displaying the color-coded image.

To start, the NumPy function **np.random.shuffle()** is used to shuffle the rows of the matrix. Next, the means of the shuffled matrix are calculated using the NumPy function **np.mean()**, with the **axis** parameter set to 1 to calculate the means for each row. The shuffled matrix is then mean-subtracted by subtracting the means from each element in the matrix. The resulting matrix is then binarized by setting any value greater than 0 to 1 and any value less than or equal to 0 to 0 using the **astype()** function.

A binary colormap displays the binarized matrix as an image using Matplotlib’s **imshow()** function. The maximum values in each row of the shuffled matrix are then calculated using **np.max()** function. The matrix is then scaled by dividing each element by its maximum value and multiplying by 255 to get the values in the range of 0 to 255. The resulting matrix is then converted to an RGB image where the green channel is set to the scaled matrix values. The image is then displayed using **plt.imshow()** function.

The signature technique is then used to recover the image clusters. Firstly, the row sums and means of the original matrix are calculated using NumPy functions **np.sum()** and **np.mean()**, respectively. The signatures are calculated by multiplying each row's sum and mean. The Similarity Matrix is then rearranged by sorting the rows according to their respective signature values using the **np.argsort()** function. The matrix is then sorted using the sorted indices, and the resulting matrix is again scaled, converted to an RGB image, and finally displayed using Matplotlib's **imshow()** function.

A picture containing chart

Description automatically generatedA picture containing shape

Description automatically generated

Chart

Description automatically generated

# Task 3:

In this task, we performed hierarchical clustering using the minimum spanning tree algorithm on a correlation matrix of the input matrix. The input matrix is first converted to a correlation matrix using the **pdist** function from the **scipy.spatial.distance** module and the threshold for edge weight is set to 0.7. Edges with weights below this threshold are removed from the correlation matrix, and the resulting matrix is converted to a binary matrix. The binary adjacency matrix is then converted to a sparse matrix using **csr\_matrix** function from the **scipy.sparse** module. The algorithm then finds clusters by iteratively finding nodes with the highest weight, and adding all neighboring nodes to the same cluster. The clusters are visualized by creating color-coded images of each cluster.

First, the correlation matrix is calculated using **squareform(pdist(matrix, metric='correlation'))**, where **pdist** is used to calculate the pairwise correlation between rows of **matrix**. The resulting matrix is then converted to an upper-triangular form since the correlation matrix is symmetric, and the upper-triangular form is sufficient for constructing the minimum spanning tree.

Next, a threshold of 0.7 is set for edge weight, and all edges with weights below this threshold are removed from the correlation matrix using **correlation\_matrix[correlation\_matrix < threshold] = 0**. The resulting matrix is converted to a binary matrix using **adjacency\_matrix = (correlation\_matrix >= threshold).astype(int)**.

The adjacency matrix is then converted to a sparse matrix using **csr\_matrix(adjacency\_matrix)**. The algorithm iteratively finds the node with the highest weight, gets its neighbors, and adds them to the same cluster. The nodes and edges from the graph are then removed, and the weights of the nodes that were removed are set to 0. This process continues until there are no more nodes with nonzero weights.

Finally, the clusters are visualized by creating color-coded images of each cluster. The input matrix is first normalized row-wise using **scaled\_matrix = matrix / max\_vals[:, np.newaxis]**, and then scaled to range between 0 and 255 using **scaled\_matrix \*= 255**. A green color-coded image of the input matrix is then created using **rgb[:, :, 1] = scaled\_matrix.astype(np.uint8)**.

The code then iterates over each cluster, creates a color-coded image of the cluster, and displays it using **plt.imshow(rgb)**. The **enumerate** function is used to add a cluster number to each output, and the indices of the rows corresponding to each cluster are printed using **print(f"Cluster {i+1}: {cluster}")**.

A picture containing chart

Description automatically generatedChart

Description automatically generated with medium confidence

Chart

Description automatically generated

Task 2 and Task 3 have different approaches to clustering and visualizing clusters.

In Task 2, the input matrix is first shuffled and mean-subtracted before being binarized and visualized as a color-coded image. The Signature technique is then used to recover image clusters, and the Similarity Matrix is rearranged by sorting the rows according to their respective signature values. The resulting matrix is again scaled, converted to an RGB image, and finally displayed as a color-coded image.

On the other hand, Task 3 involves hierarchical clustering using the minimum spanning tree algorithm on a correlation matrix of the input matrix. The correlation matrix is first converted to a binary matrix, and the adjacency matrix is then converted to a sparse matrix. The algorithm then finds clusters by iteratively finding nodes with the highest weight and adding all neighboring nodes to the same cluster. The clusters are visualized by creating color-coded images of each cluster.

Both tasks involve clustering and visualizing clusters using color-coded images, but Task 2 uses the Signature technique while Task 3 uses the minimum spanning tree algorithm on a correlation matrix. The approaches are different, but both can be useful depending on the specific data and problem at hand.