MINI-PROJECT 2 – PROJECT REPORT
CSE 564 – VISUALISATION AND VISUAL ANALYTICS
Aakash Bhatia (SBU ID: 112685865)

Video: https://www.youtube.com/watch?v=YW9F8ye68xo
Dataset: University Rankings 2020
Source: Kaggle (https://www.kaggle.com/joeshamen/world-university-rankings-2020)
Dimensions: (1397 x 16)

Task 1:

I have sampled my data create 2 datasets.

1) Random sampling: I randomly sampled by data to retain only 25% of the dataset. On starting up my Flask server, the full dataset is randomly sampled and stored in a file 'random_sample.csv'. I performed this sampling using in the function 'create_random_samples()'. The function utilizes df.sample(frac = 0.25) command which takes a pandas dataframe and samples 25% of the data.

```
def create_random_samples():
    global df1
    df_random = df1.sample(frac=0.25)
    df_random.to_csv("random_sample.csv", index=False)
```

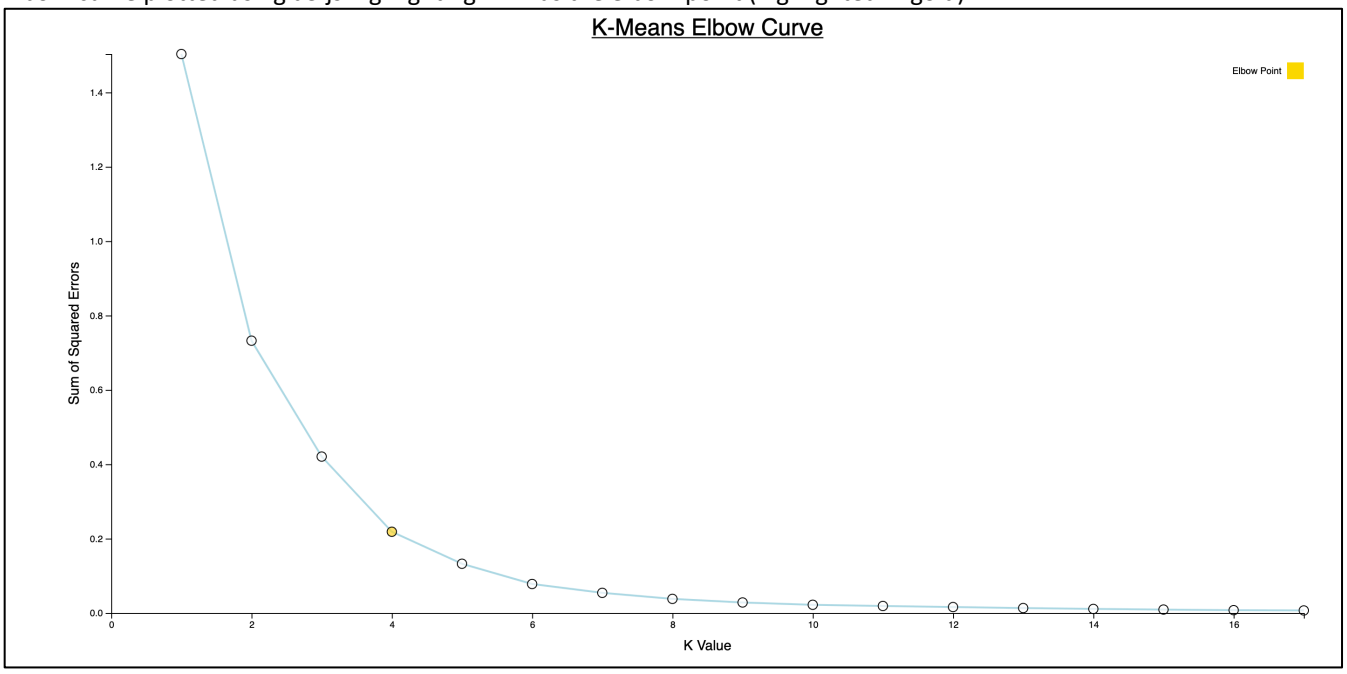
2) Random stratified sampling: I used the k-means algorithm to cluster my dataset. I sampled 25% of the data from each cluster and appended the data to create a new dataset. On starting up my Flask server, the full dataset is sampled using the method described above and stored in a file 'random_stratified.csv'. I performed this sampling using in the function 'create_random_stratified_samples()'.

   To obtain the appropriate k value to perform the k-means algorithm, I ran the k-means algorithm for k = 1 to k = 17. I noticed that the curve flattens at approximately k = 9. The elbow point for the dataset is at k = 4. Hence, I used k = 4 in my k-means implementation. The elbow curve for the same is plotted using d3.js in the front-end of my application.

   Running k-means from k=1 to k = 17

```
def create_random_stratified_samples():

    global df1

    #array to store sum of squared distances
    sum_of_squared_distances = []
    #array to store k values
    k_vals = []
    #array to store elbow values
    elbow_values = []

    #performing k-means for k = 1 to k = 17
    for k in range(1,18):
        #appending k value in k_vals array
        k_vals.append(k)
        #running k-means function
        k_means = KMeans(n_clusters=k)
        #fitting the values and appending them to sum_of_squared_distances array
        model = k_means.fit(df1.loc[:, :'Overall_Ranking'])
        sum_of_squared_distances.append(k_means.inertia_ * 10**-12)

    #transforming the data to be able to send it to front end
    elbow_values.append(k_vals)
    elbow_values.append(sum_of_squared_distances)
    final_elbow_vals = pd.DataFrame(elbow_values)
    final_elbow_values = final_elbow_vals.T
    final_elbow_values.columns = ['K', 'Value']
    mapping = final_elbow_values.to_dict(orient="records")
    chart_data = json.dumps(mapping, indent=2)
    data = {'chart_data': chart_data}
```

Elbow-curve plotted using d3.js highlighting k = 4 as the elbow-point (highlighted in gold)



Running k-means for k = 4

```
k_means = KMeans(n_clusters=4)
model = k_means.fit(df1.loc[:, :'Overall_Ranking'])
y = k_means.predict(df1.loc[:, :'Overall_Ranking'])
df1['Cluster'] = y
```

Task 2:

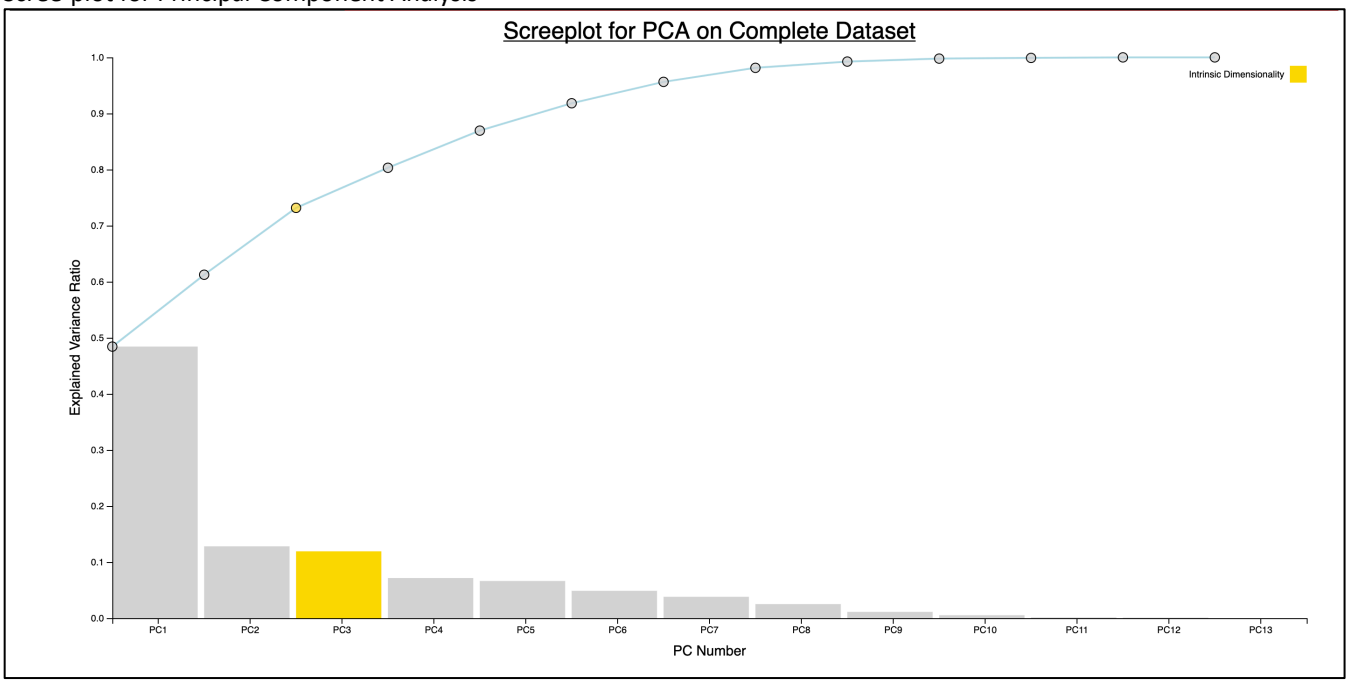
Performing PCA to obtain the intrinsic dimensionality.

1) I performed Principal component analysis on each dataset to obtain the intrinsic dimensionality. I performed the following steps:
    a. Ran the PCA algorithm for n = 13 to obtain the eigen values for each PC.
    b. Plotted a scree-plot to visualize the values obtained. The scree-plot contains the following details:
        i. A bar graph to visualize the eigen values for each PC.
        ii. A line graph to show the total contribution to the dataset considering multiple PC's.
        iii. Both these provide complimentary data and best describes the contribution of each PC to the respective datasets.
    c. To highlight the intrinsic dimensionality, I considered the PC values that contributed <80% of the data.
    d. The intrinsic dimensionality changes each time the dataset is sampled. I noticed the intrinsic dimensionality for the full dataset is PC3. However, the value varies between PC3 and PC4 for the randomly sampled and stratified sampled datasets.
    e. The scree-plot is visualized using d3.js in the front end of the application. The intrinsic dimensionality is highlighted in gold.
    f. The intrinsic dimensionality is calculated in the function 'calculate_intrinsic_dimensionality(df)' which takes a dataframe and computes the intrinsic dimensionality using the process described above.

Performing PCA for n_components = 13

```
def calculate_intrinsic_dimensionality(df):
    #variables to obtain required values to send to front-end. Count_2 holds the intrinsic dimensionality
    count = 0
    count_2 = 0
    sum = 0
    sum_2 = 0
    ptg_data = []
    list_intrinsic_dimensions = []

    #performing PCA = 13
    pca = PCA(n_components=13)

    #scaling the data using standardscalar()
    x = StandardScaler().fit_transform(df.loc[:, :'Overall_Ranking'])
    principal_components = pca.fit_transform(x)
    data = pca.explained_variance_ratio_
    data = data.tolist()
```

Scree-plot for Principal Component Analysis



2) To obtain the attributes with the highest PCA loadings, I performed the following steps:
   a. I used the intrinsic dimensionality obtained from point 2.1 to run PCA on each dataset.
   b. I obtained the sum of squared values for each attribute loading.
   c. I considered the top 3 values as the attributes as the attributes with highest PCA loading.
   d. I have implemented this in the function calculate_top_attributes() which takes the dataframe, like of PC's part of the intrinsic dimensionality and the intrinsic dimensionality the dataset. Using the intrinsic dimensionality, it computes the attributes with highest PCA loadings and returns the values.
   e. The code snippet below explains the process carried out

Code to calculate the 3 attributes with highest PCA loadings

```python
def calculate_top_attributes(df, list_intrinsic_dimensions, count):
    df_temp = df.loc[:, :'Overall_Ranking']
    list_country = df['Country'].tolist()
    list_uni = df['University'].tolist()
    list_rank = df['Rank'].tolist()

    #running PCA for the intrinsic dimensionality
    pca = PCA(n_components=count)

    #transforming the values using standardscalar()
    x = StandardScaler().fit_transform(df_temp)
    pc1 = pca.fit_transform(x)
    data = pca.explained_variance_ratio_
    data = data.tolist()

    #obtaining the loadings per column and storing them in a dataframe
    loadings = pd.DataFrame(pca.components_.T, columns= list_intrinsic_dimensions,
                            index=df_temp.columns)

    #calculating the values with the highest PCA loadings
    for i in loadings:
        loadings[i] = loadings[i] ** 2
    loadings['sum'] = loadings.sum(axis=1)
    df_final = df_temp.loc[:, loadings.nlargest(3, ['sum']).index.tolist()]
    df_final['Country'] = list_country
    df_final['University'] = list_uni
    df_final['Rank'] = list_rank

    #formatting data to send to front end
    mapping = df_final.to_dict(orient="records")
    chart_data = json.dumps(mapping, indent=2)
    data = {'chart_data': chart_data}
    return data
```

Task 3:

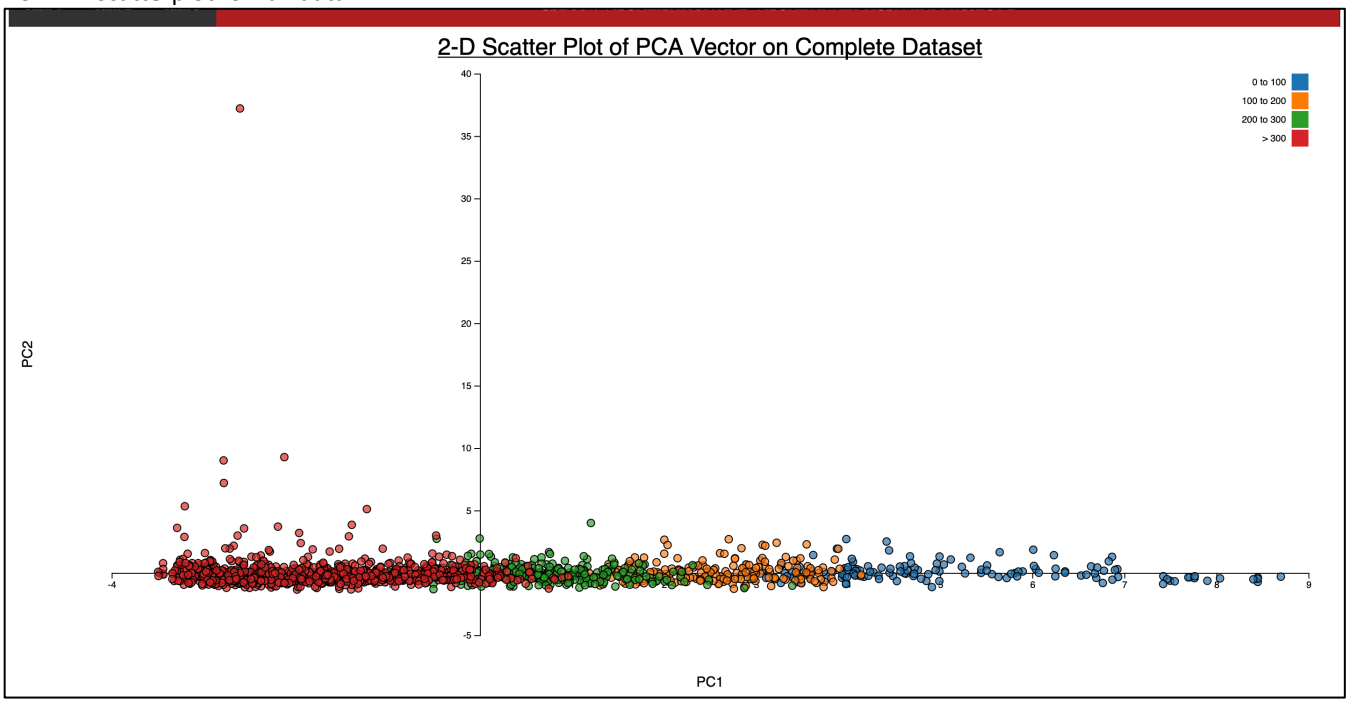
Note: For each of the below points, I have used the University Rank Bracket (0 to 100, 100 to 200, 200 to 300 and >300) as my labels.

1) 2-D scatter plots for the top 2 PCA vectors.
   To visualize this, I performed the following steps:
   a) I ran PCA with n_components = 2
   b) I appended the University Name, Country and Rank bracket the university belongs to for each row item.
   c) The top two PCA vectors is calculated in the function calculate_top_PC_vals() which takes a dataframe as input and returns the top 2 PCA vectors.
   d) The data is visualized using a 2-D scatterplot for the full data, randomly sampled data and random stratified data.
   e) Code snippet below explains the process followed:

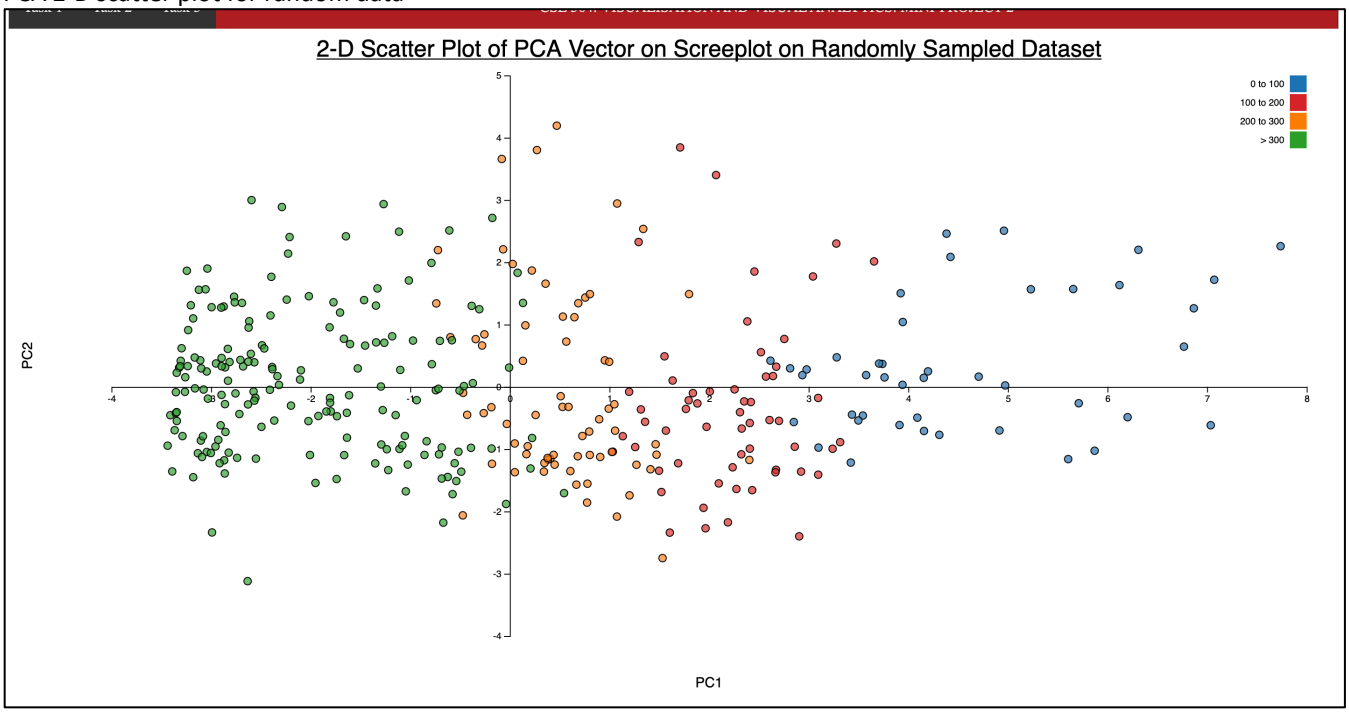
   Code snippet to obtain top 2 PCA vectors

```python
def calculate_top_PC_Vals(df):

    #Performing PCA for n=2
    pca = PCA(n_components=2)

    #Scaling data using Standard Scalar and appending the University, Country and Rank
    x = StandardScaler().fit_transform(df.loc[:, :'Overall_Ranking'])
    pc = pca.fit_transform(x)
    data = pca.explained_variance_ratio_
    principal_Df = pd.DataFrame(data=pc, columns=['PC1', 'PC2'])
    principal_Df['University'] = df['University']
    principal_Df['Country'] = df['Country']
    principal_Df['Rank'] = df['Rank']

    #Formatting the data to send to the front end
    mapping = principal_Df.to_dict(orient="records")
    chart_data = json.dumps(mapping, indent=2)
    data = {'chart_data': chart_data}
    return data
```

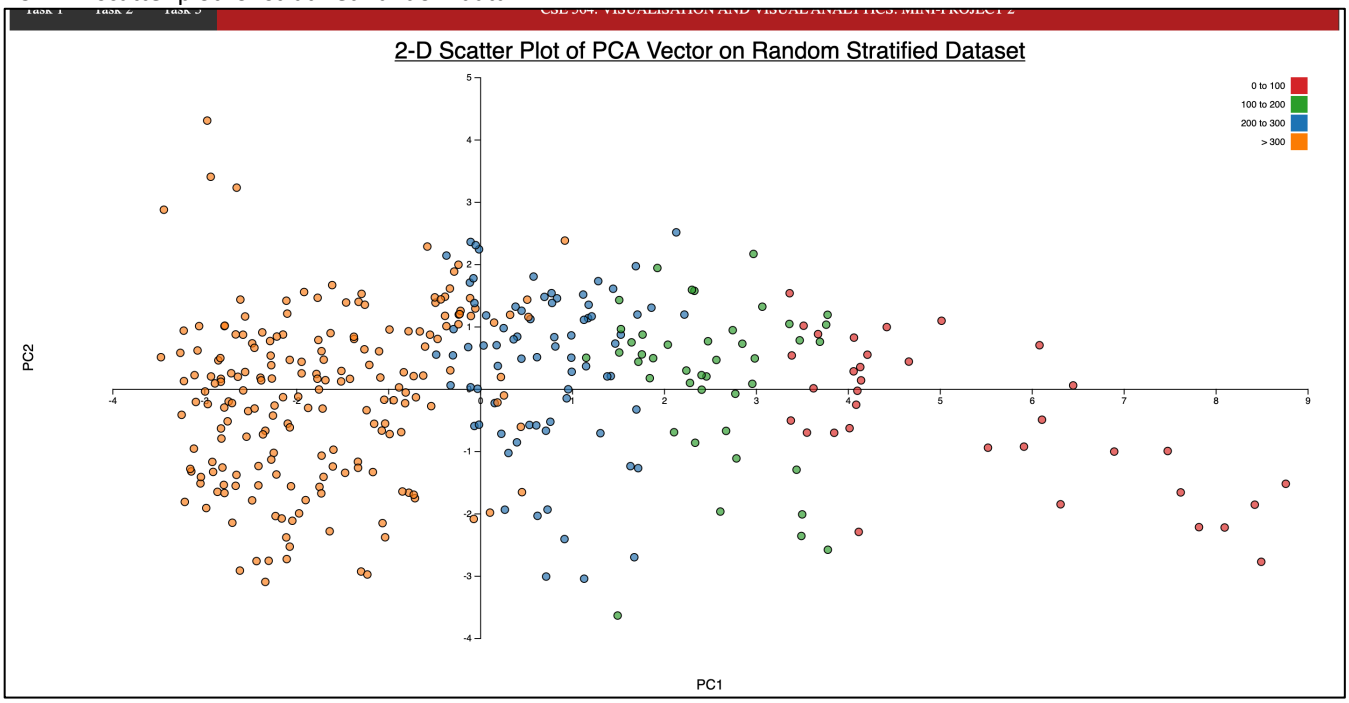
PCA 2-D scatterplot for full data



PCA 2-D scatter plot for random data



PCA 2-D scatter plot for stratified random data



2) MDS (Euclidean and Correlation)

To implement the MDS algorithm, I performed the following steps:

a) For each dataset, I first computed the pairwise distance. The pairwise_distance function takes “metric = euclidean” and “metric = correlation” as input. Hence, this allowed me to obtain the matrix required for the MDS algorithm for both Euclidean and Correlation.

b) I ran MDS with n_components = 2, dissimilarity = precomputed and n_jobs = -1.

c) I appended the University Name, Country and Rank bracket the university belongs to for each row item.

d) Visualized the data using a 2-D scatterplot for the full data, randomly sampled data and random stratified data.

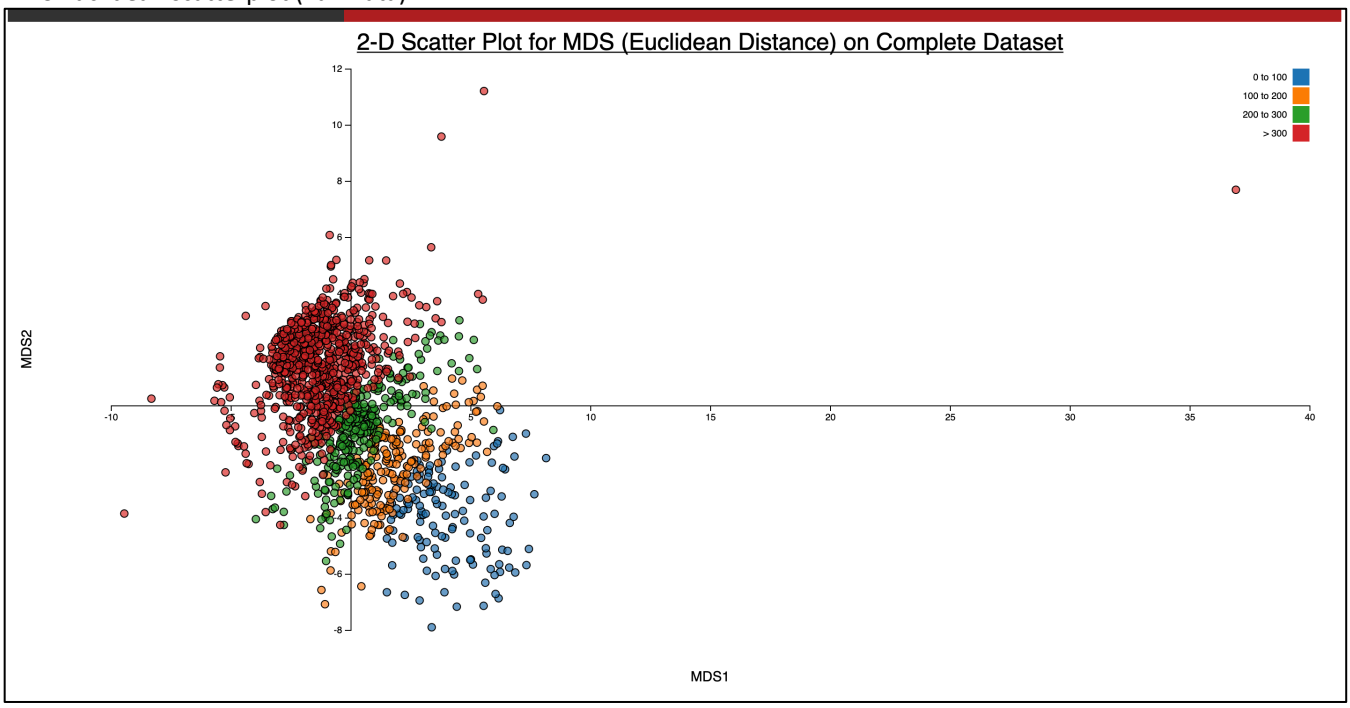
e) The code snippets shows how this was performed

Code snippet for MDS implementation

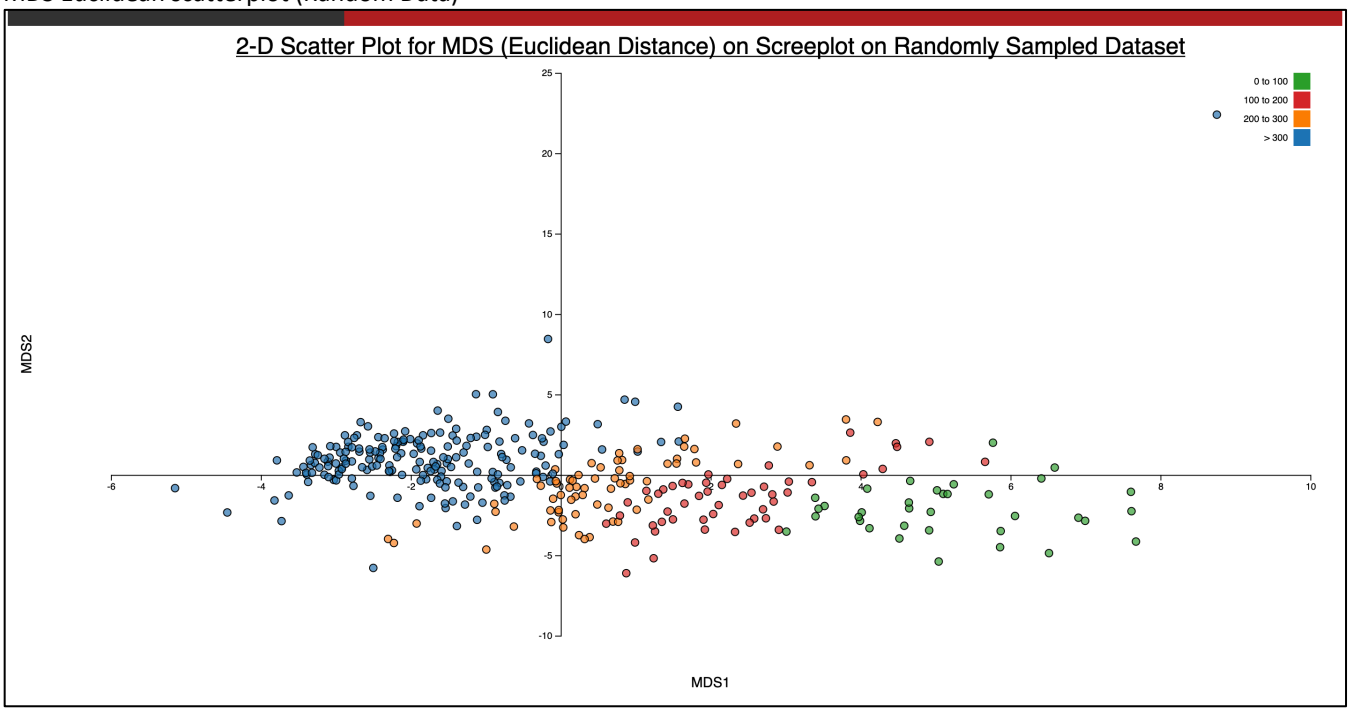
```
def calculate_mds(df, distance):
    df_temp = df.loc[:, :'Overall_Ranking']

    #Scale data
    df_scaled = StandardScaler().fit_transform(df_temp)
    df_scaled_final = pd.DataFrame(df_scaled)

    #compute required pairwise distance
    matrix = metrics.pairwise.pairwise_distances(df_scaled_final, metric=distance)
    list_country = df['Country'].tolist()
    list_uni = df['University'].tolist()
    list_rank = df['Rank'].tolist()
```

```
#perform MDS
mds = MDS(n_components=2, dissimilarity='precomputed', n_jobs=-1)

values = mds.fit_transform(matrix)

df_final = pd.DataFrame(values, columns= ['MDS1', 'MDS2'])
df_final['Country'] = list_country
df_final['University'] = list_uni
df_final['Rank'] = list_rank

mapping = df_final.to_dict(orient="records")
chart_data = json.dumps(mapping, indent=2)
data = {'chart_data': chart_data}
return data
```

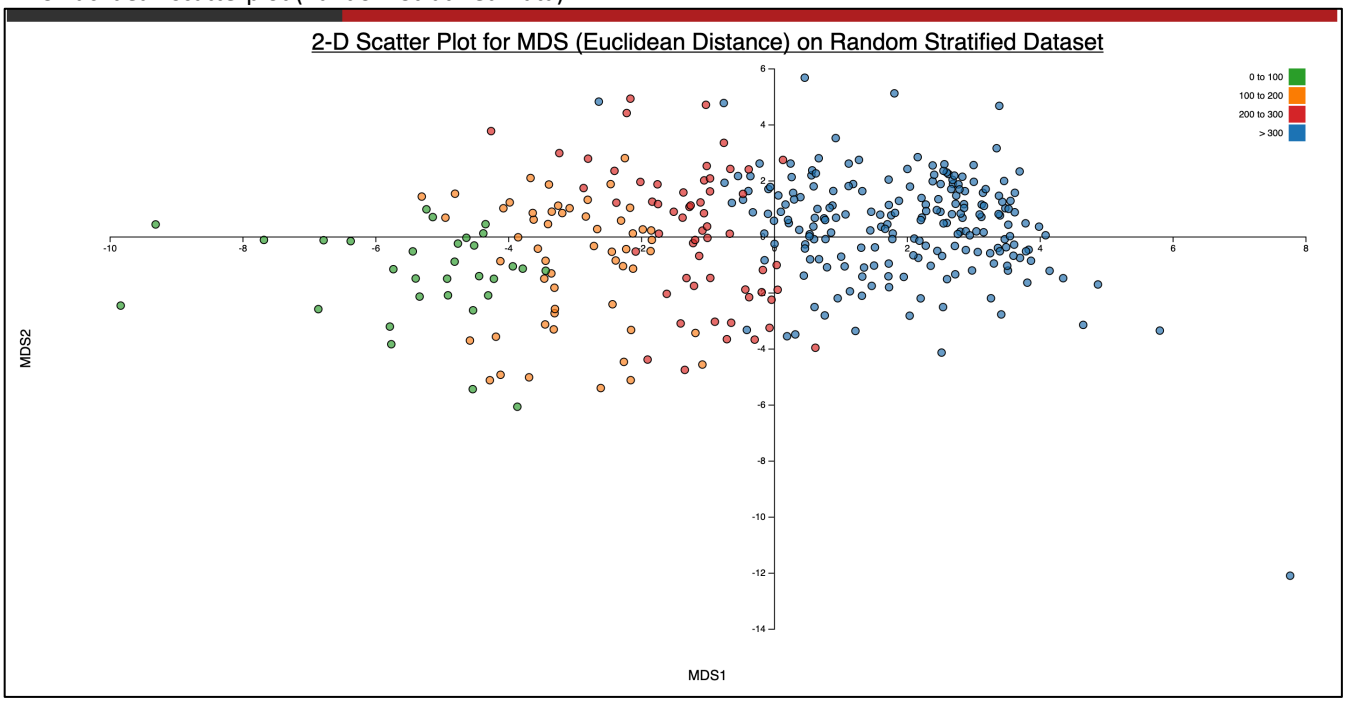
MDS Euclidean scatterplot (Full Data)



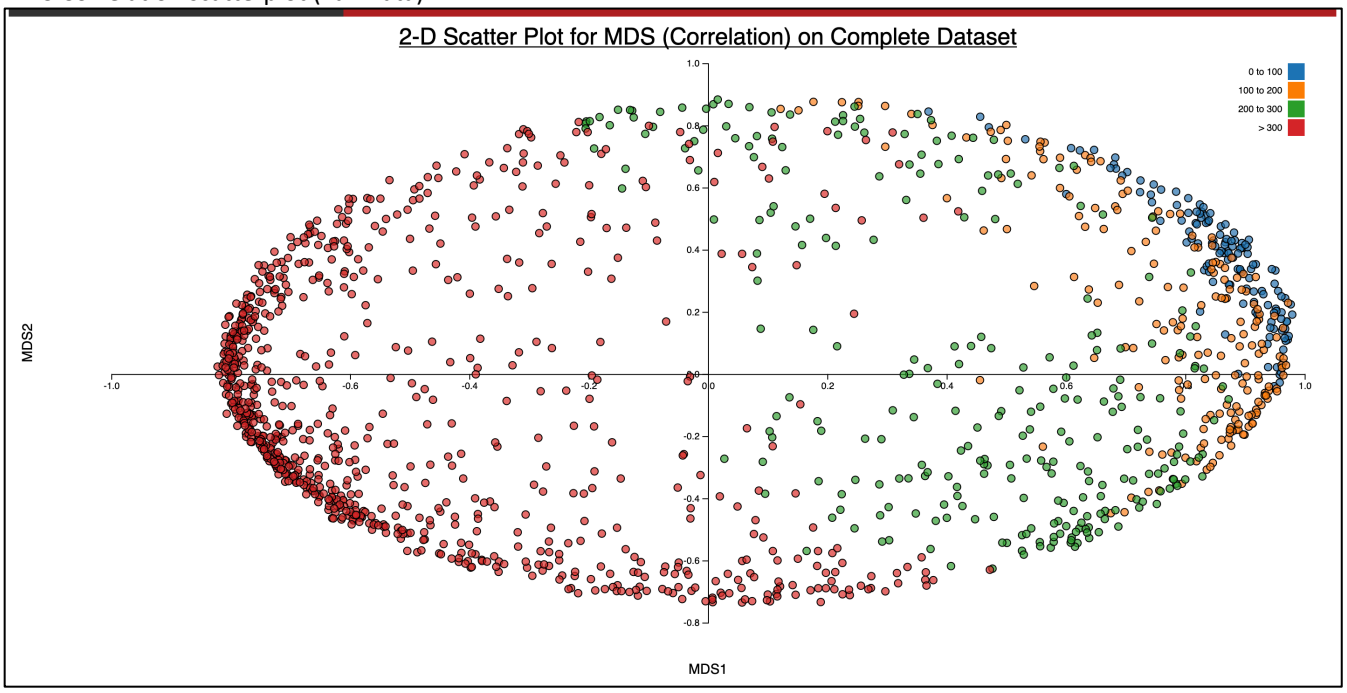
MDS Euclidean scatterplot (Random Data)



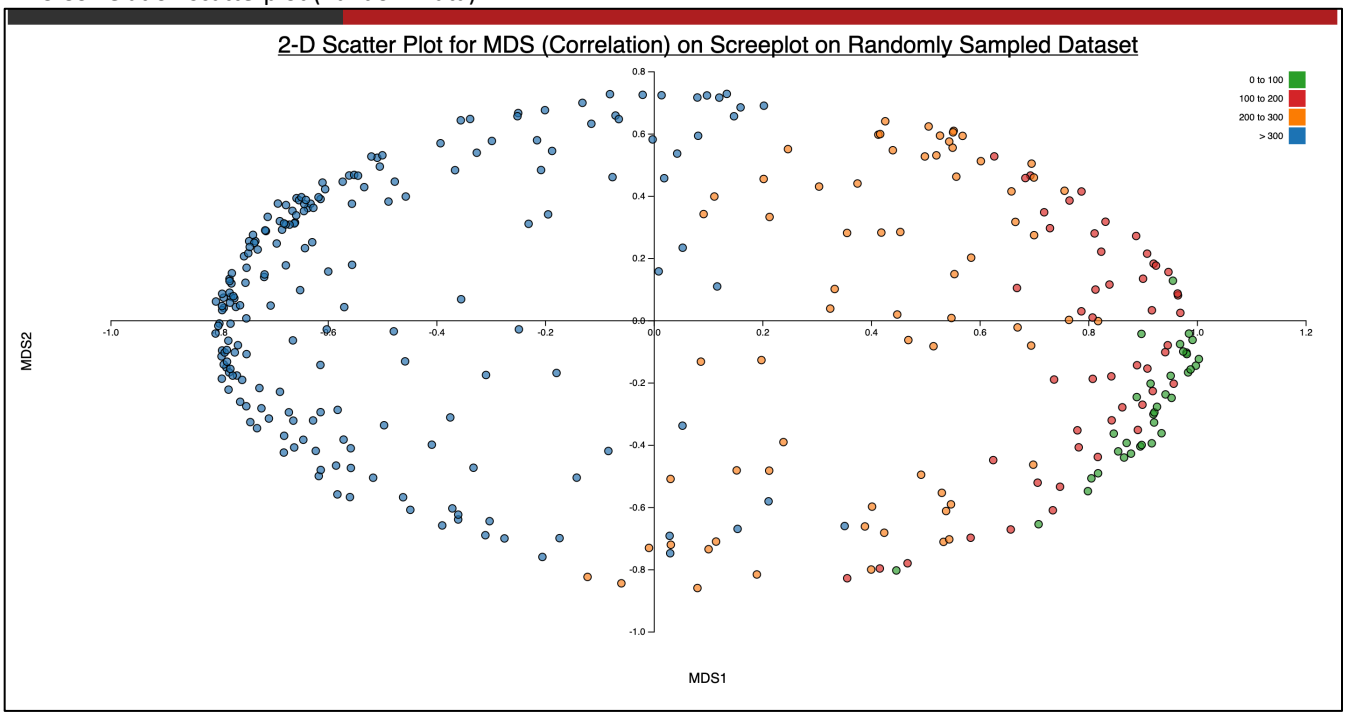
MDS Euclidean scatterplot (Random Stratified Data)



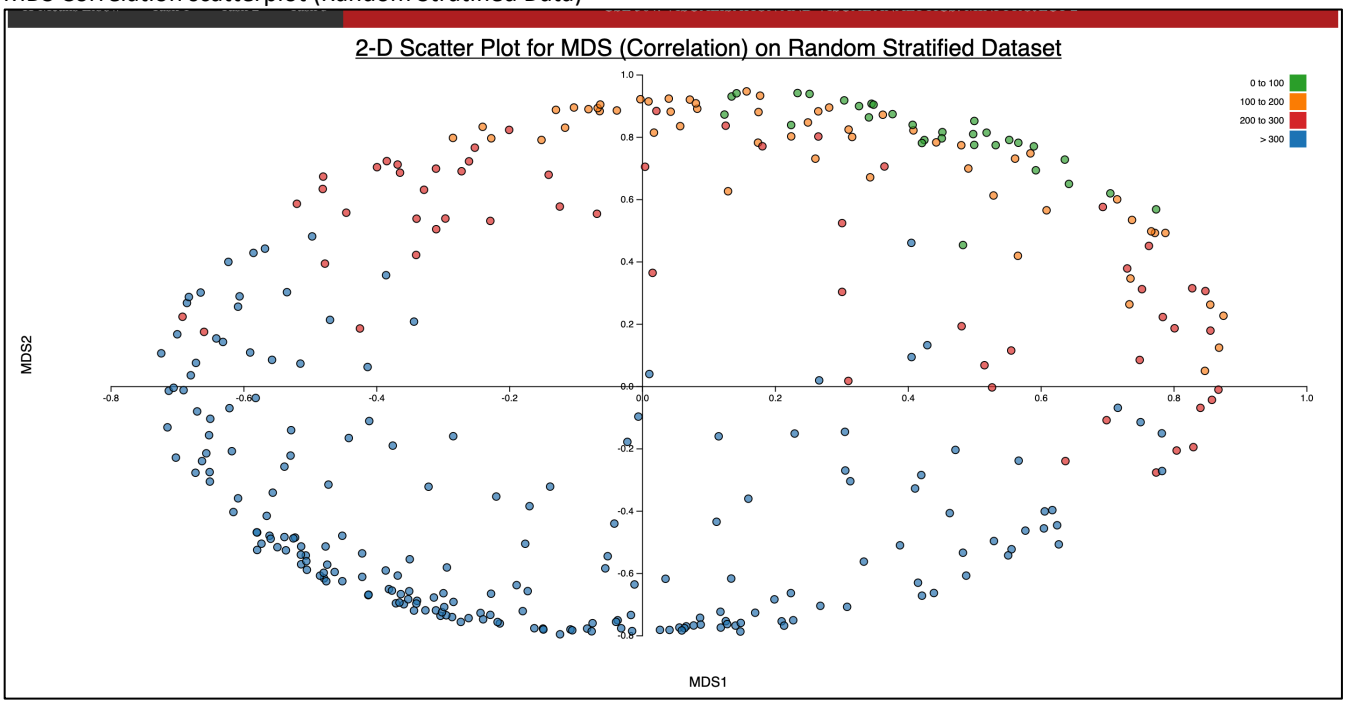
MDS Correlation scatterplot (Full Data)



MDS Correlation scatterplot (Random Data)



MDS Correlation scatterplot (Random Stratified Data)



3) Scatterplot Matrix for 3 attributes with highest PCA loadings

To create the scatterplot matrix, I performed the following steps:

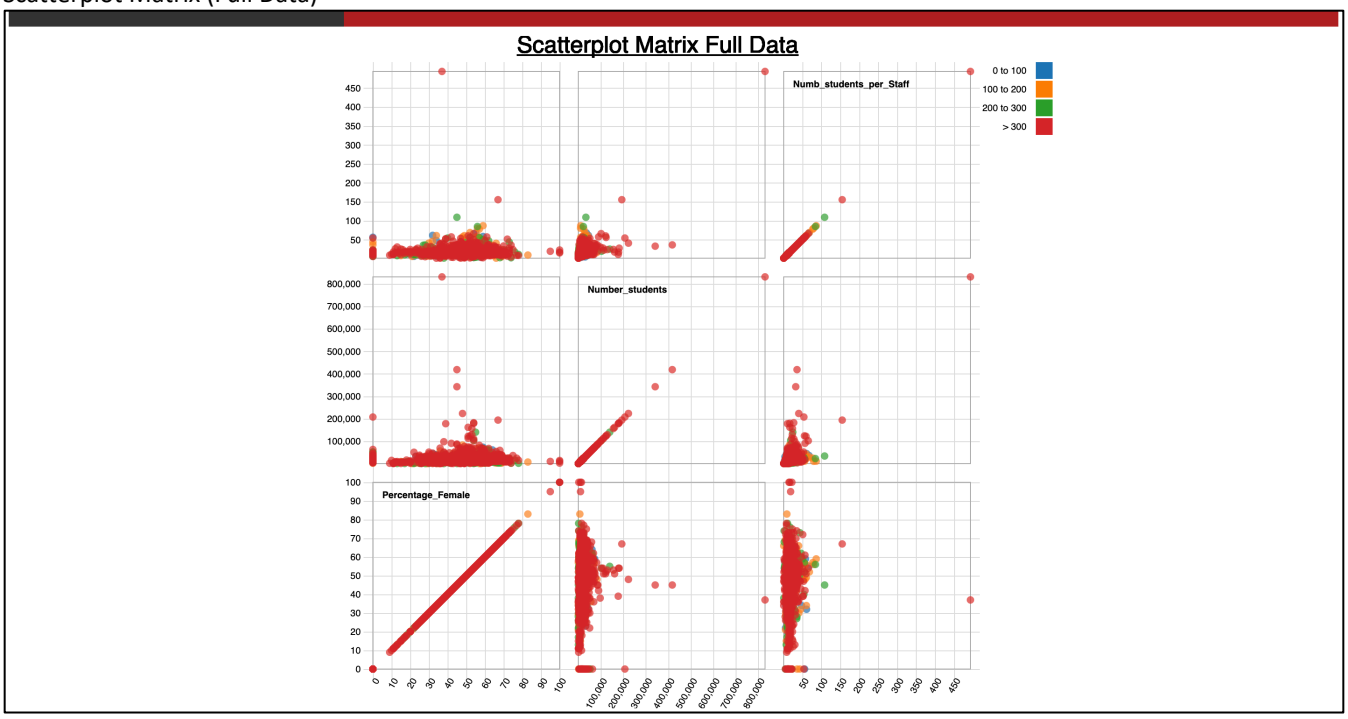
a) I obtained the 3 attributes with the highest PCA loadings as mentioned in Task 2.2.

b) I created a scatterplot matrix for the values obtained for each dataset.

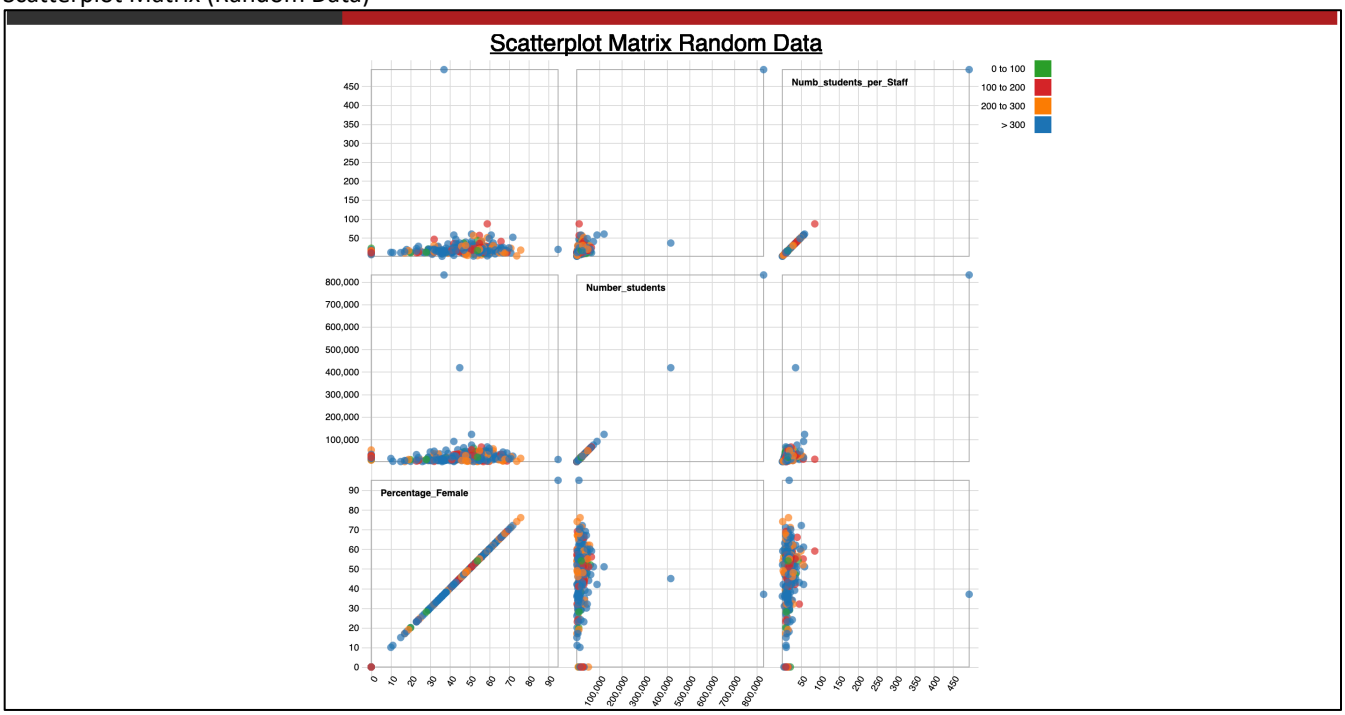
c) Since the values for each dataset is different, I dynamically obtained the column names from the json data sent to the front end using d3.keys() in d3.js. (Code snippet below)

```
//obtaining unique column names
var column_names = {},
    columns = d3.keys(data[0]).filter(function(d) { return (d !== "Country" && d!=='University' && d!=='Rank'); }),
    n = columns.length;

//appending each column name to column_names
columns.forEach(function(columns) {
column_names[columns] = d3.extent(data, function(d) { return d[columns]; });
});
```

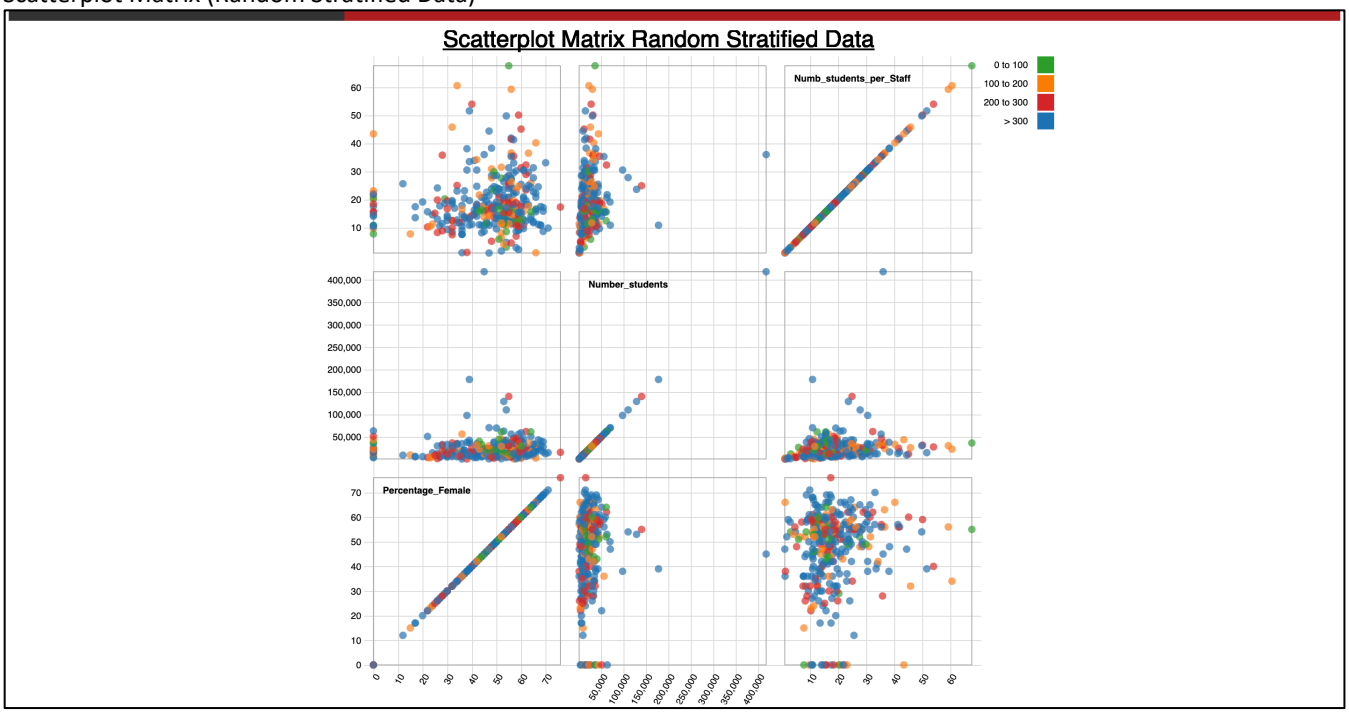
Scatterplot Matrix (Full Data)



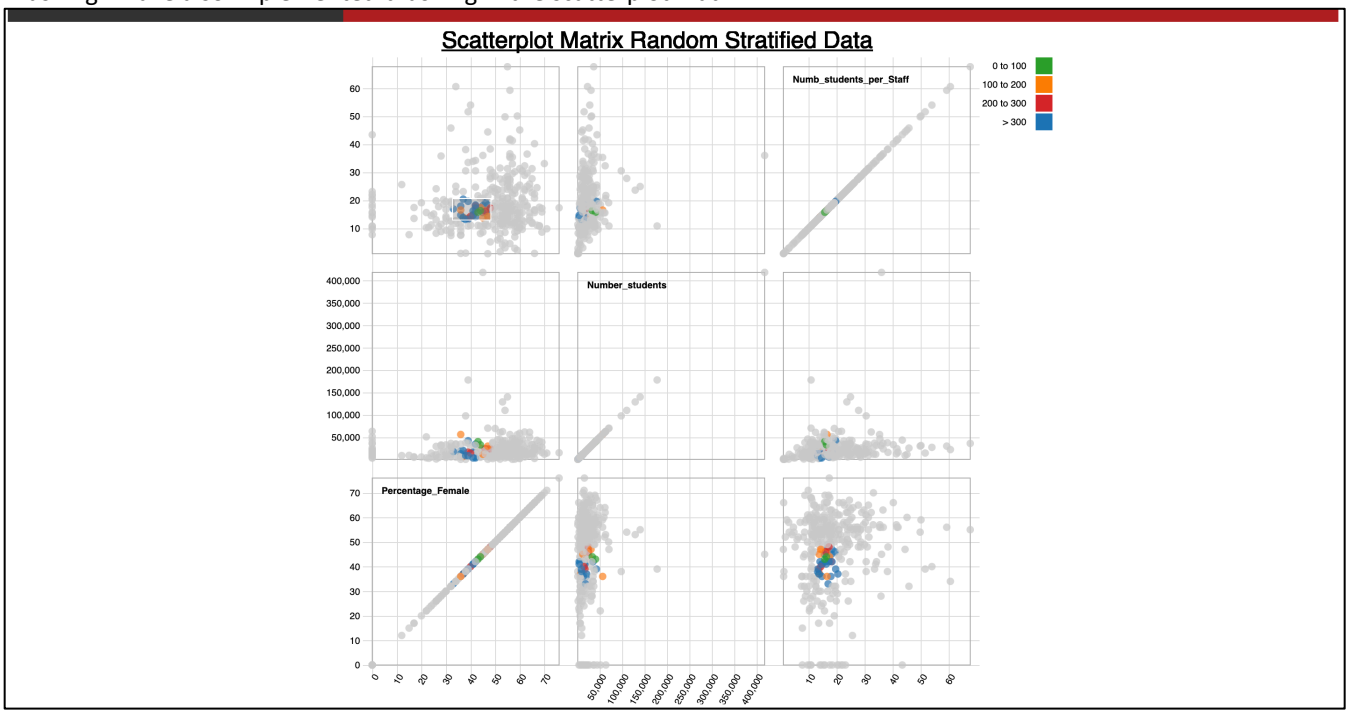
Scatterplot Matrix (Random Data)



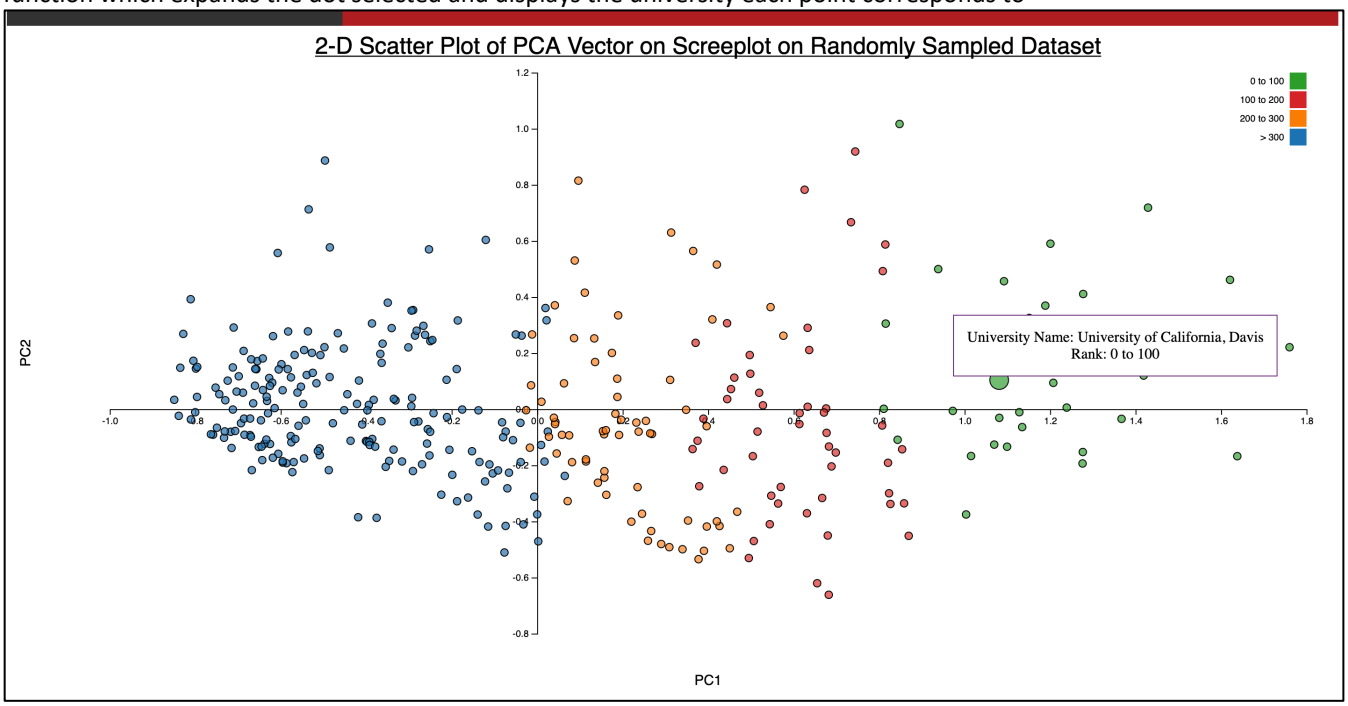
Scatterplot Matrix (Random Stratified Data)



Brushing: I have also implemented brushing in the scatterplot matrix



Mouseover: Since all my visualization are based on the University ranking bracket, I have also implemented a mouseover function which expands the dot selected and displays the university each point corresponds to



Logic used for data-refresh:

1) Throughout the application described above, I am using flask to refresh the data in the front end. Each time the user needs to change the graph, the data is refreshed using flask and the new plot is generated using d3.js. I have performed this by using different paths in my flask server.

   Code snippet

```
@app.route("/", methods = ['GET', 'POST'])
def index():
    return render_template("index.html", data = "")

@app.route("/elbow-k-means", methods = ['POST'])
def get_elbow():
    global elbow_vals
    return jsonify(elbow_vals)

@app.route("/screeplot_full_data", methods = ['POST'])
def scree_plot_full_data():
    global df1
    data, list_intrinsic_dimensions, count_2 = calculate_intrinsic_dimensionality(df1)
    return jsonify(data)
```

   As you can see above, each activity (elbow-k-means, screeplot_full_data and so on) are different paths defined in the backend. In the frontend, I call the function in javaScript which will perform the following tasks:
   - Take the path as an input parameter
   - Refresh the data in the frontend by carrying out the functions defined within this path
   - Display the plot in the frontend based on the new data.

   I implemented the data-refresh logic in this manner to make sure that the latest data is being used to create the plots. Hence, if the data changes, the graph changes accordingly.

Observations:

1) The full data and data obtained by random stratified sampling have a similar distribution. Hence, the graphs obtained for these two datasets are usually similar. However, this is not always the case with the randomly sampled dataset
2) I noticed the MDS algorithm provided more consolidated view of the data with close but distinct clusters. PCA on the other hand provided a scatterplot which was elongated and spread out.
3) On performing MDS (Euclidean) on the full data I noticed the scatterplot provides very distinct and easily identifiable clusters compared to PCA.
4) On running PCA and MDS algorithms on the both the sampled datasets, I noticed that the scatter plot was somewhat of a 'zoomed in' version of the full data plot. The values on the x axis of the full dataset are (-5, -10, -15…) whereas the values in the sampled dataset are (-1, -2, -3, …) and hence is a closer more detailed view.
5) Using the full data and randomly stratified data, the columns selected for the scatterplot matrix were the same (order might be different, but the columns were the same). However, this isn't always the case with the randomly sampled data.
6) The shape of the visualization provided by the MDS algorithm using correlation is very interesting. We can clearly see universities ranked 0 to 100 on one end of the ellipse and >300 on the other end. However, neighboring clusters were not separated well.

Submitted Code Files:

- Folder structure:

```
Code/
        flaskDirectory/
                templates/
                        index.html
        app.py
        data_full.csv
```

- Steps to run code.
  a. Store the "code" folder
  b. Navigate to the code/flaskDirectory/
  c. "python app.py" OR "python3 app.py"