Visualization Assignment - 2

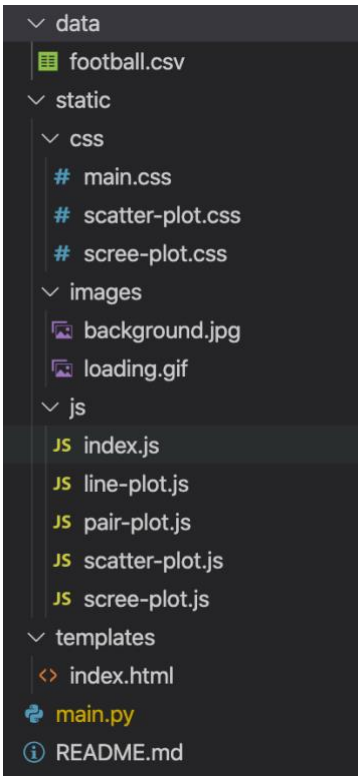
7th April 2020

Implementation Details

I have designed a responsive user interface using Bootstrap and jQuery. To plot the charts, I have used D3.js. The backend is handled by python (Flask). I have performed code modularization so that new pieces can be added by making only a few changes.

Please find the Link to YouTube.

Overall Code Structure



Data Set

I have taken FIFA 19 dataset with shape (2855, 27). The entire dataset can be found here.

Tools Used

Libraries/Technologies used:

- D3.js v5
- Bootstrap and jQuery
- HTML, CSS and JS
- Python with Pandas, Scikit and Numpy libraries
- Flask libraries to set up the server

Tasks

1) Data Clustering and Decimation

Implement Random Sampling

```
def generateRandomSample():
    global randomSample
    chosen_idx = np.random.choice(sampleSize, replace=False, size=sampleSize)
    randomSample = specs_csv.iloc[chosen_idx]
```

The code snippet above randomly chooses sampleSize (calculated based on the size of the original dataset) indices and saves those rows into another dataframe.

Implement Stratified Sampling

```
def clusteringForK():
    k = 4
    kmeans = KMeans(n_clusters=k)
    kmeans.fit(specs_csv)
    labels = kmeans.labels_
    specs_csv['clusterId'] = pd.Series(labels)
```

To implement stratified sampling K Means clustering was performed on the data.
K is chosen to be 4 as most of the data can be effectively clustered into 4 clusters.

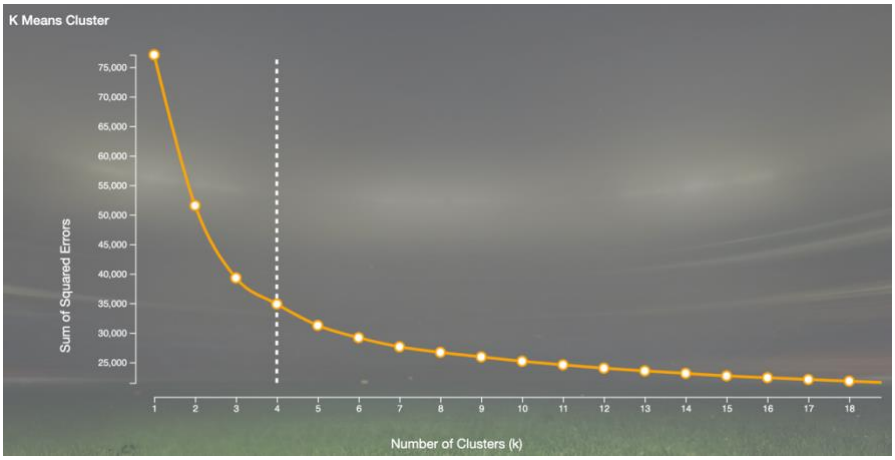
```
def generateAdaptiveSample():
    global adaptiveSample

    stratifiedCluster = {}
    lenData = len(specs_csv)

    for i in range(4):
        cluster = specs_csv[specs_csv['clusterId'] == i]
        clusterSize = len(cluster) * sampleSize / lenData
        chosen_idx = np.random.choice(
            int(clusterSize), replace=False, size=int(clusterSize))
        stratifiedCluster[i] = cluster.iloc[chosen_idx]

    for i in range(4):
        adaptiveSample = adaptiveSample.append(stratifiedCluster[i])
```

Code to perform stratified sampling. Random rows from each stratum is chosen based on the proportion of the sample in that strata.



The result of the K-Means clustering is presented as a line plot. The code involved in building the line plot is a part of line-plot.js and involves utilizing the circle and line svgs.
Observation: Consider 4 to be the ideal number of clusters.

2) Dimension Reduction

Finding and marking Intrinsic Dimensionality of the data using PCA and listing out the features with top PCA loadings

```
def performPCA(data, numberOfPC=15):
    # Standardize the data
    X_std = StandardScaler().fit_transform(data)
    # Create a PCA instance
    pca = PCA(n_components=numberOfPC)
    principalComponents = pca.fit_transform(X_std)
    return pca.explained_variance_ratio_, pca.explained_variance_ratio_.cumsum()
```

The steps in the above code have to be performed to obtain the explained variance and its cumulative sum.

```
@app.route("/plot_scree")
def plot_scree():
    sourceArray = {}
    pcaCumSum = []
    pca = []
    for i in range(0, 3):
        source, name = dataSourceMapping(i)
        try:
            pca, pcaCumSum = performPCA(source)
        except:
            e = sys.exc_info()[0]
            print(e)

        sig = {}
        squaredLoadings = sqaureLoadings[name]

        i = 0
        for col in mainColumns:
            sig[col] = squaredLoadings[i]
            i = i+1

        sourceArray[name] = {'pca': pca.tolist(),
                             'pcaCumSum': pcaCumSum.tolist(),
                             'significance': sig
                             }

    return json.dumps(sourceArray)
```

This is then sent to the front end of the application as JSON when the application makes a request.

```
for i in range(0, 3):
    sig = {}
    source, name = dataSourceMapping(i)
    sl = loadSquareLoadings(source, 10)
    sqaureLoadings[name] = sl
    i = 0
    for col in mainColumns:
        sig[col] = sl[i]
        i = i+1
    topFeatures[name] = []
    for w in sorted(sig, key=sig.get, reverse=True):
        topFeatures[name].append(w)
```

```
def loadSquareLoadings(data, dimen):
    eigenValues, eigenVectors = generateEigenValues(data)
    squaredLoadings = []

    featureCount = len(eigenVectors)
    for fId in range(0, featureCount):
        loadings = 0
        for compId in range(0, dimen):
            loadings = loadings + \
                eigenVectors[compId][fId] * eigenVectors[compId][fId]
        squaredLoadings.append(loadings)
    return squaredLoadings
```

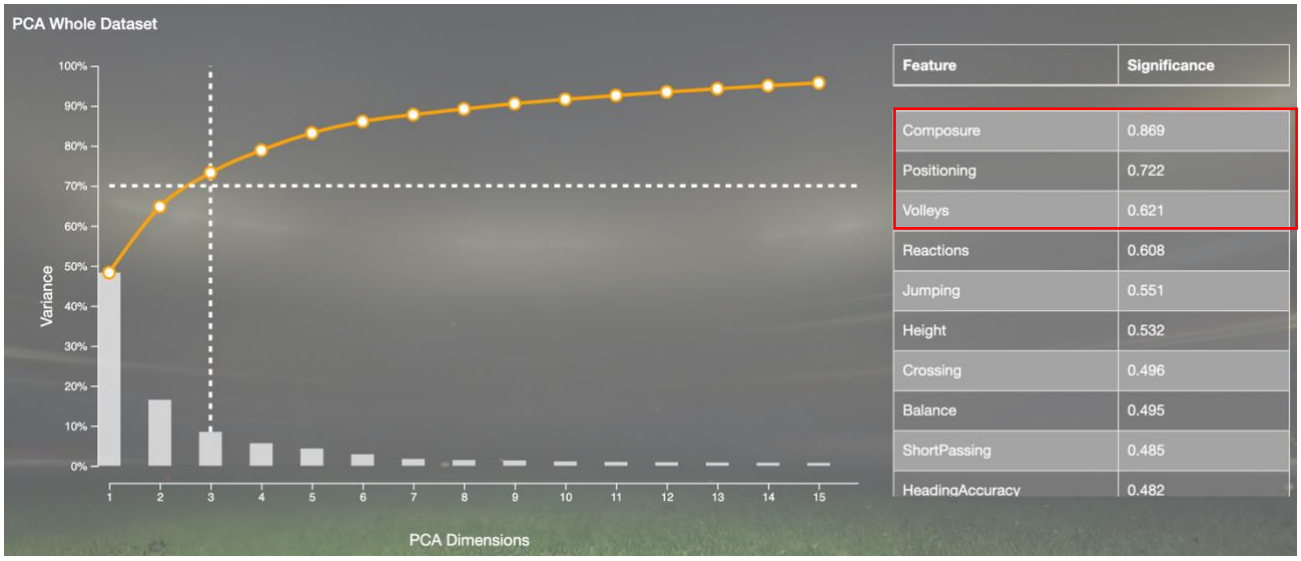
Function defined to calculate the significance by adding square loadings for each feature in the data set.

```
def generateEigenValues(data):
    covMat = np.cov(data.T)
    eigenValues, eigenVectors = np.linalg.eig(covMat)

    # Sort Eigen Values
    ev = eigenValues.argsort()[::-1]
    eigenValues = eigenValues[ev]
    eigenVectors = eigenVectors[:, ev]
    return eigenValues, eigenVectors
```

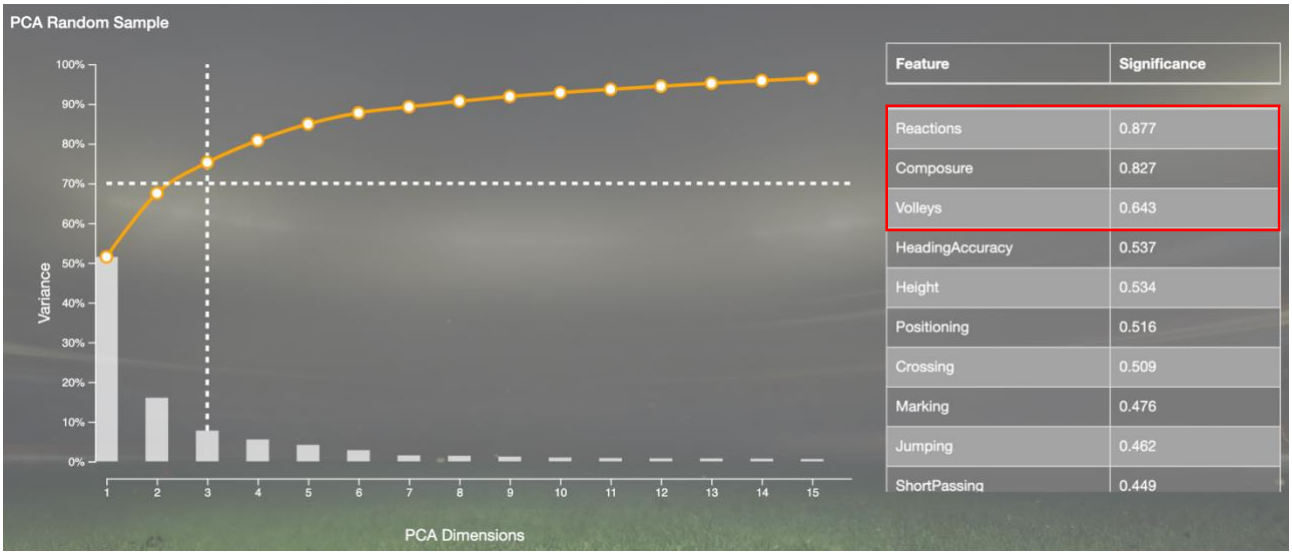
The eigen vector and eigen values are generated using the numpy linear algebra library.

PCA for the different datasets and PCA Loadings



Feature	Significance
Composure	0.869
Positioning	0.722
Volleys	0.621
Reactions	0.608
Jumping	0.551
Height	0.532
Crossing	0.496
Balance	0.495
ShortPassing	0.485
HeadingAccuracy	0.482

Observation: 3 is the number of dimensions that can explain 70% of the variance in the data.



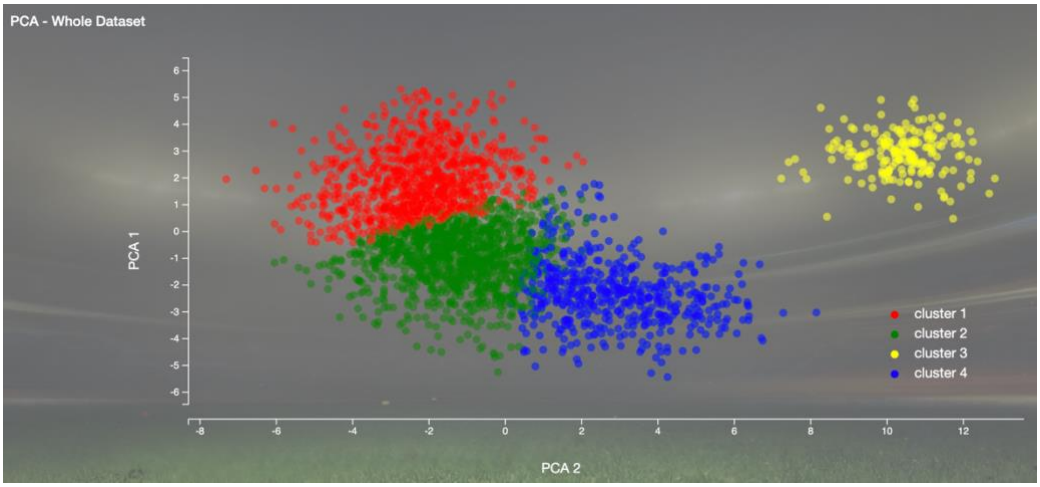
Observation: As stated above, 3 is the intrinsic dimensionality of both the datasets.

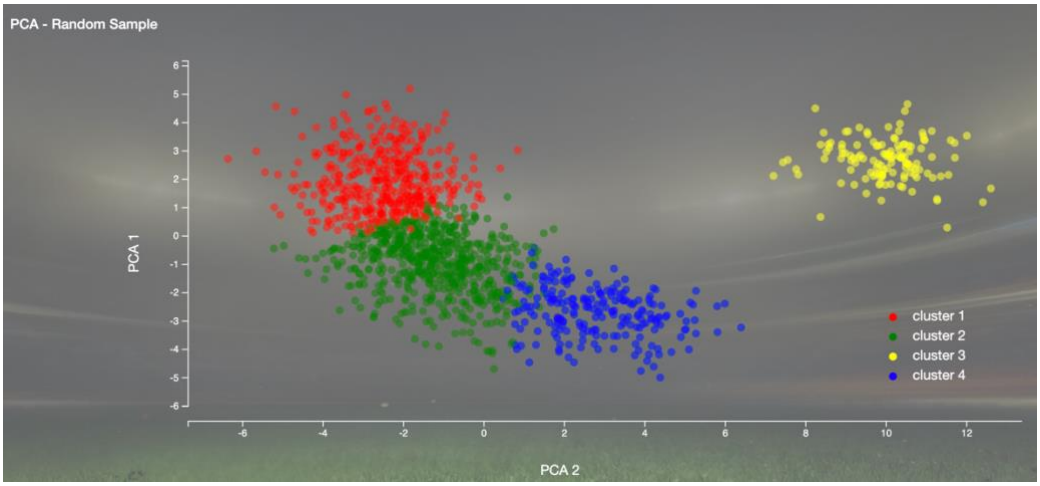
The code to plot scree plot in d3 is a part of scree-plot.js.

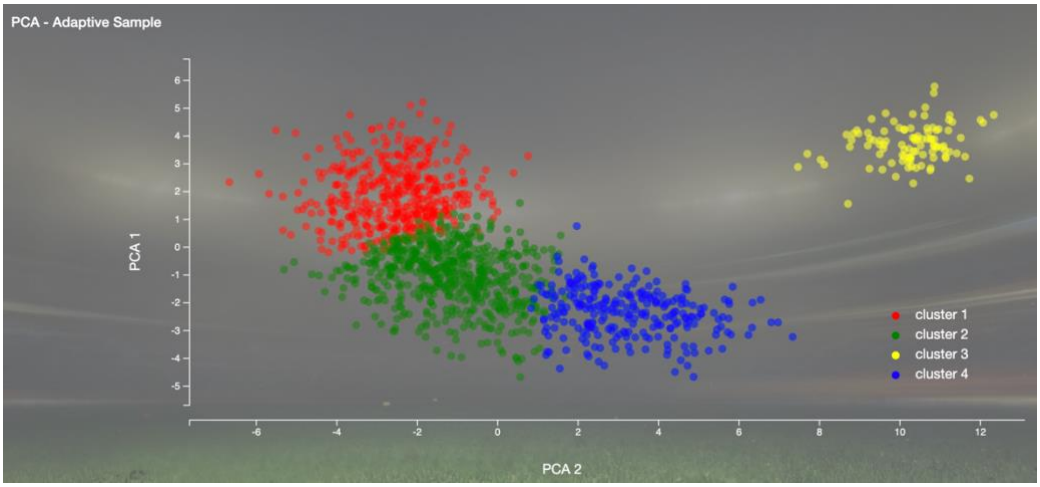
3 Other Visualizations

Scatter Chart for PCA1 vs PCA2

Similar procedure of performing PCA as before. However, the number of dimensions is reduced to just 2. The result of the above operation is plotted as a scatter chart. The d3 code for the scatter chart is a part of scatter-plot.js.

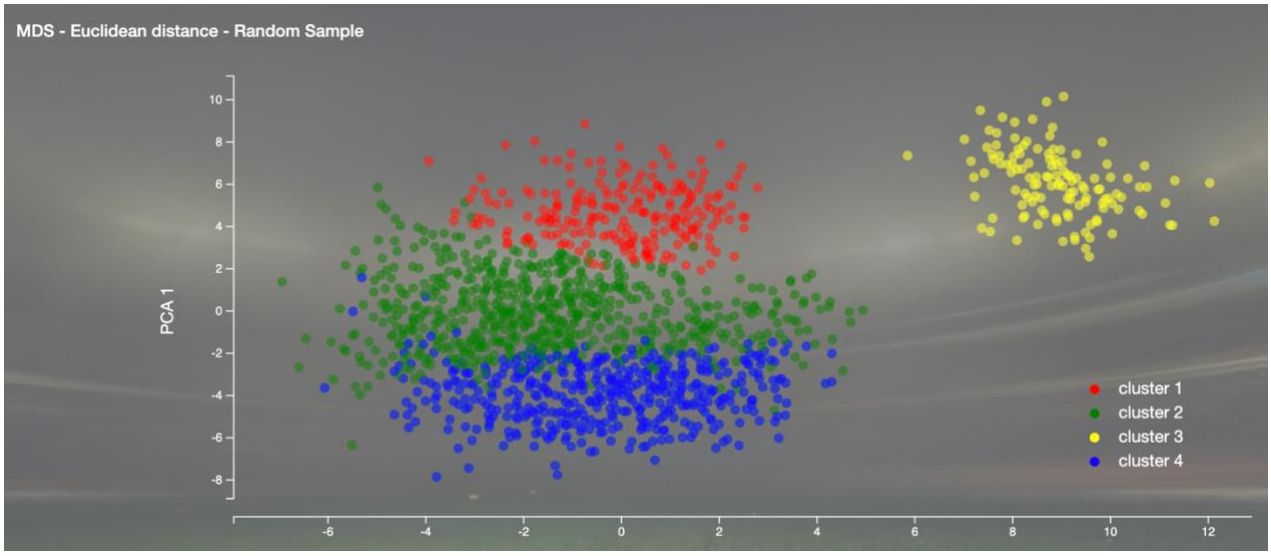


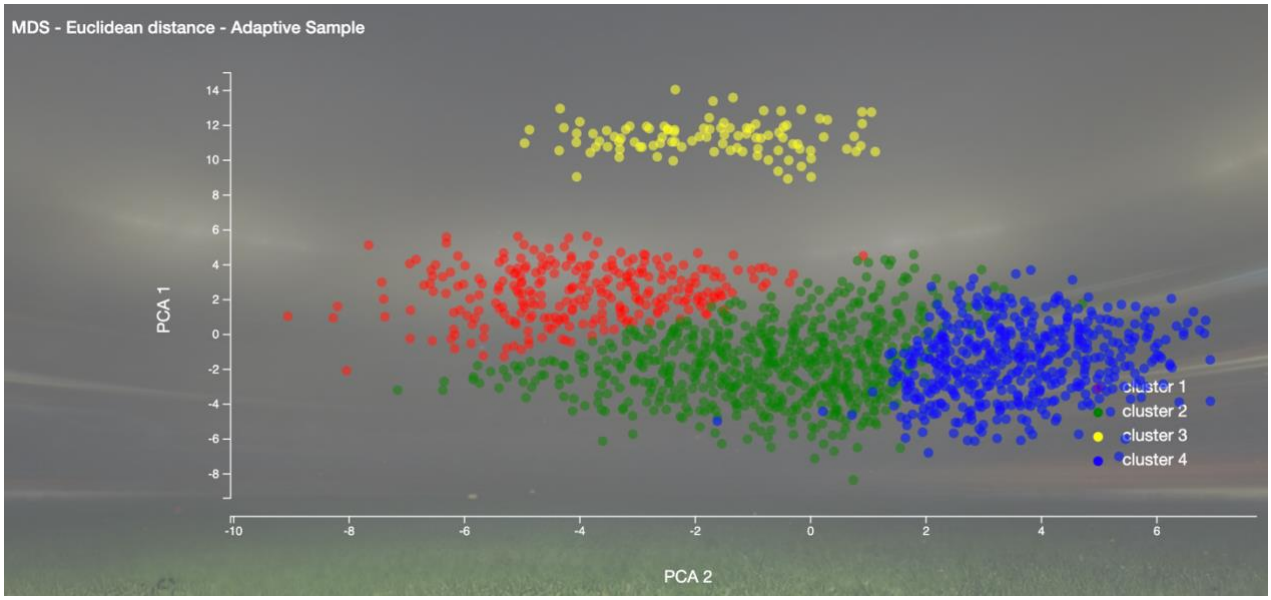




Observation: We observe clear distinction of clusters in the complete sample, random sample or adaptive sample. The spread of the data can be observed as well.

Euclidean Distance for PCA1 vs PCA2



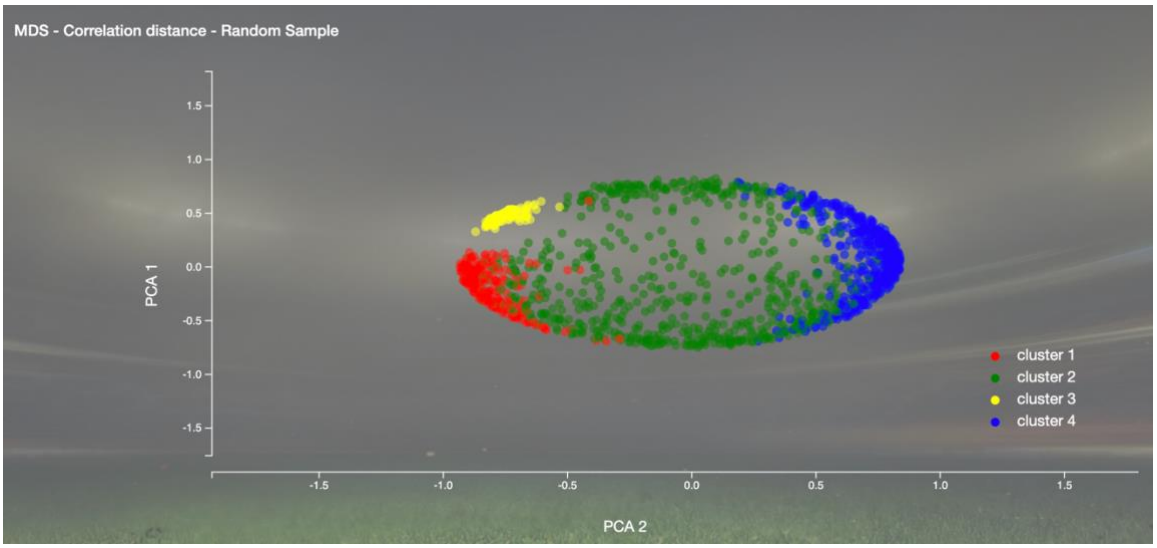


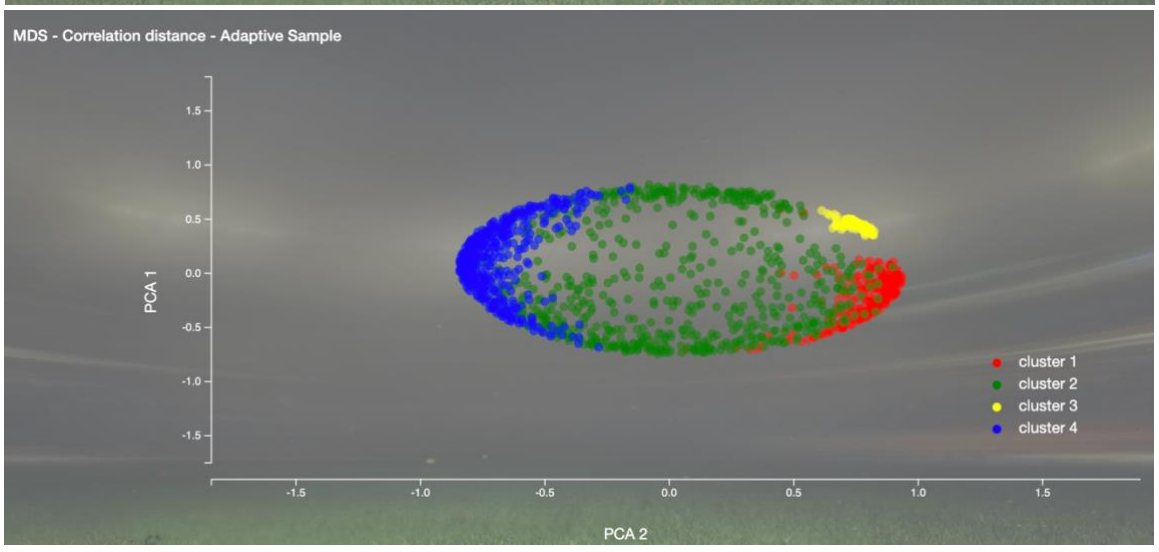
```
def euclidean_mds(t):
    source, name = dataSourceMapping(t)
    sourceC, nameC = dataSourceMapping(t, False)
    X_std = StandardScaler().fit_transform(source)
    model = MDS(n_components=2, random_state=1)
    res = model.fit_transform(X_std)
    df = pd.DataFrame(res)
    df['cluster'] = sourceC['clusterId']
    df['name'] = copy_specs_csv['Name']
    return df
```

Function to generate Euclidean distance based MDS values.

Observation: As stated above there is a clear distinction amongst the different clusters.

Correlation Distance for PCA1 vs PCA2



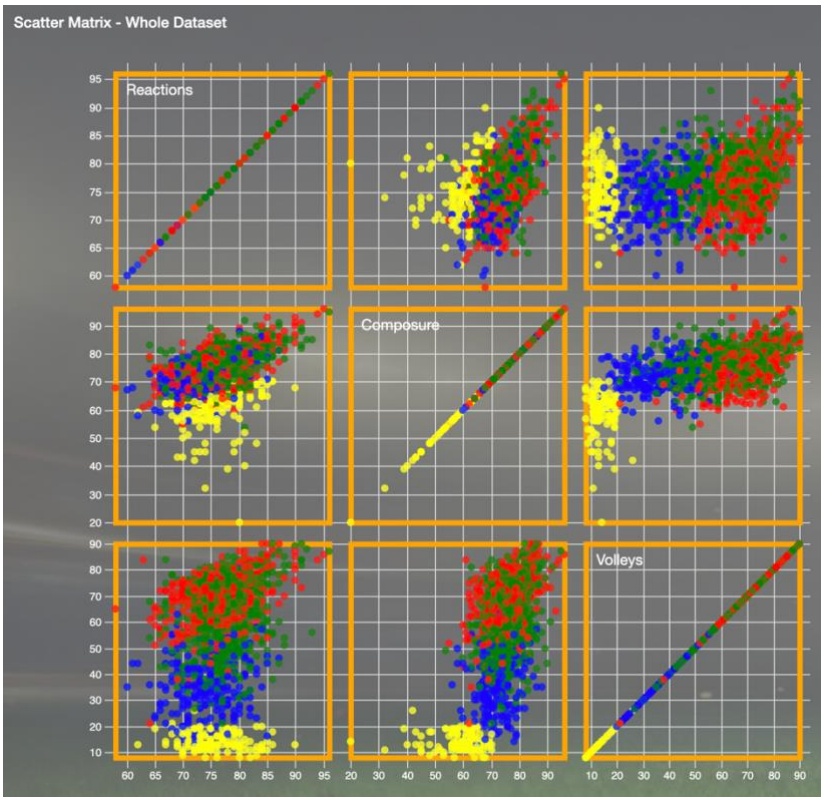


```
def correlation_mds(t):
    source, name = dataSourceMapping(t)
    sourceC, nameC = dataSourceMapping(t, False)
    X_std = StandardScaler().fit_transform(source)
    Y = cdist(X_std, X_std, 'correlation')
    model = MDS(n_components=2, random_state=1,
        dissimilarity='precomputed', n_jobs=-1)
    res = model.fit_transform(Y)
    df = pd.DataFrame(res)
    df['cluster'] = sourceC['clusterId']
    df['name'] = copy_specs_csv['Name']
    return df
```

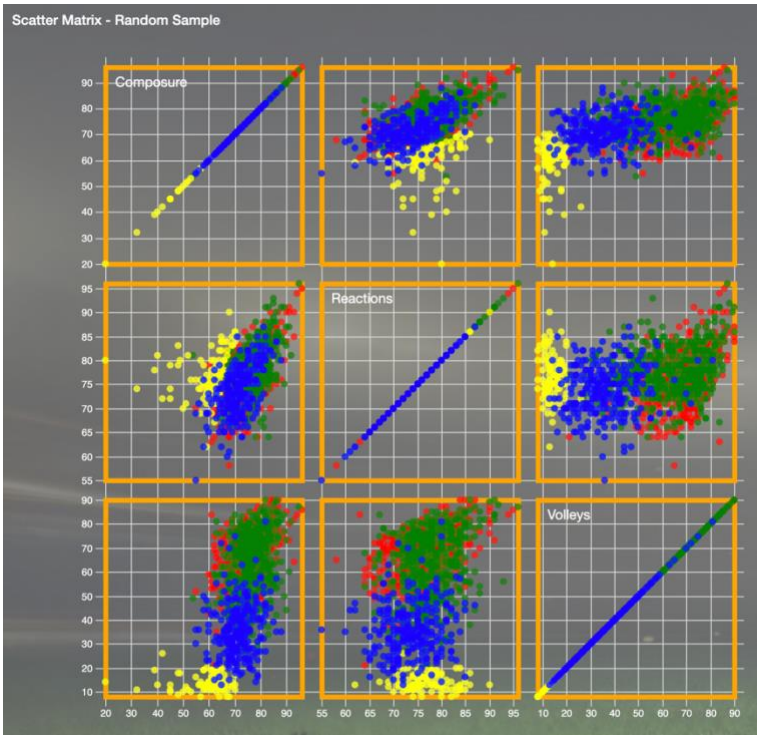
Observation: There is a clear distinction amongst the different clusters.

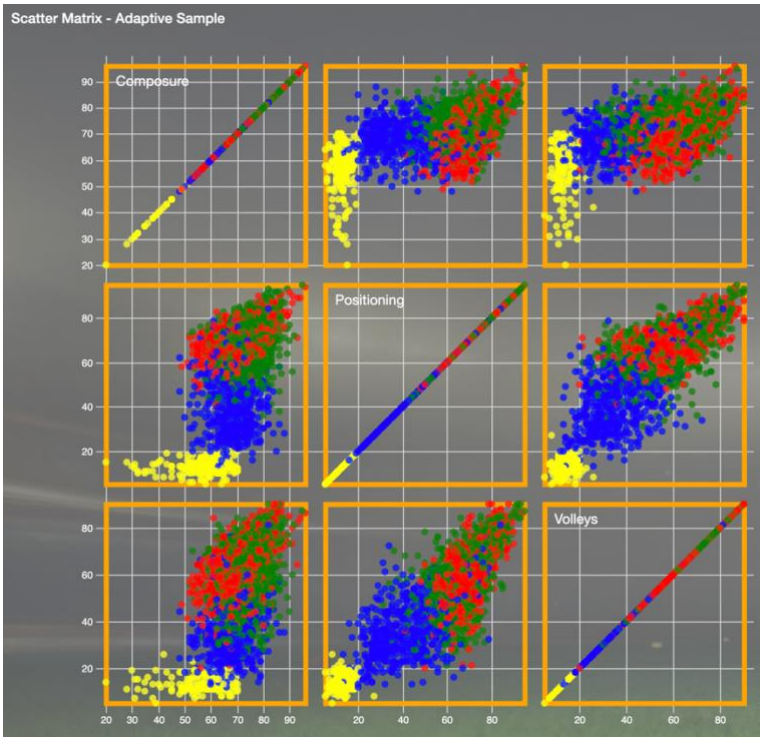
The proximity of the points belonging to a cluster In MDS indicates the effective clustering.

Scatter Matrix – Topmost significant Components



Observation: For all the datasets the three most significant features are plotted against each other. There are a few linear dependencies but mostly there is no distinct pattern with any pair of features. The cluster distinction is somewhat clear. There is a good spread of data.


Scatter Matrix - Random Sample
Composure
Reactions
Volleys


Scatter Matrix - Adaptive Sample
Composure
Positioning
Volleys