Visualization Assignment-2

April 2, 2017

The d3 based visual interface displays the statistical analysis of growth of cancer in various people. The visual layout is created using a client-server architecture with python as the backend and d3.js for the front end. Flask server on python is used to host the web-application and visualization was done on top of a bootstrap dashboard. The data from python to d3 is transferred in json format. Url mappings are done on the python side code and the d3 visualization is done using the data gathered from these URL mappings.

Preparing the Data.

The interface is built on top of the data available in cancer_data.csv file. The file contains details of diagnosis results for different cancer patients. The data set was obtained from kaggle and contains 570 data points and around 35 attributes. While reading the file from mongo db, the data set is scaled down to 11 dimensions. The data is read by the python program from the csv file and is stored in mongo db for persistent storage. When the UI is rendered, the data is fetched from Mongo db and processed according to the various tasks . The processed data is then passed onto the UI d3 layer for rendering various elements.

Loaded data to mongo db as :

C:\MongoDB\bin\mongoimport.exe -d vis2 -c cancerdata --type csv --headerline --file cancer-data.csv

The data from Mongo db is fetched in python . First the connection is established with the mongo db installed locally and the data loaded using the above command is identified using the collection name – ‘cancer data’. The below method in python is used to fetch the data in this collection.

```
def get_data():
    connection = MongoClient(MONGODB_HOST, MONGODB_PORT)
    collection = connection[DBS_NAME][COLLECTION_NAME]
    cancerdata = collection.find(projection=FIELDS)
    json_cancerdata = []
    for d in cancerdata:
        json_cancerdata.append(d)
        connection.close()
    return json_cancerdata
```

The result from the mongo db is in json format, which is converted to a 2D array format for easier processing in python. The below method does this. The FIELD object given in the below method specifies the 11 dimensions to which I’m scaling the data.

```
def get_2d_data():
    json_cancerdata = get_data()
    result = []
    for data in json_cancerdata:
        temp =[]
        for test in FIELD:
            temp.append(data[test])
```

```
        result.append(temp)

    return result
```

The file script.js contains the d3 based java script code that renders the page. The script is coupled with the python server and different areas in the layout is rendered using the processed data that is passed from python .

Requirements

1. Random sampling and stratified sampling/K-means clustering and elbow curve.

The filtered data (2D data) as obtained in the above steps is used for sampling. For stratified sampling first K-means clustering is performed on the data to form clusters and the sample data is taken from these clusters randomly. Hence the samples formed after the stratified sampling process will contain data from the different clusters, based on the size of each cluster.K-means clustering is performed in python using the kmeans api in the sklearn library . Code snippet is given below

Python code.

```
def plot_kmeans():
    global d_data #get the 2D data.
    # perform k means clustering with k varying from 1 to 15.
    kMeansVar = [KMeans(n_clusters=k).fit(d_data) for k in range(1, 15)]
    centroids = [X.cluster_centers_ for X in kMeansVar]
    #find the centroids of each clusters and compute the means squared error
    k_euclid = [cdist(d_data, cent) for cent in centroids]
    dist = [np.min(ke, axis=1) for ke in k_euclid]
    #Pass the data to d3, to plot the elbow curve using different values of K and
their errors
    wcss = [sum(d**2) for d in dist]
    plotwcss = {"0": wcss}
    plotwcss = json.dumps(plotwcss, default=json_util.default)
    return plotwcss
```

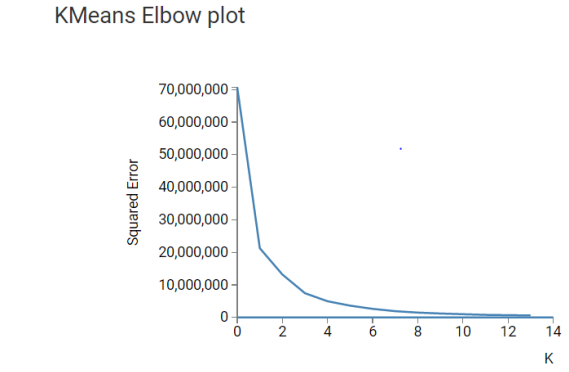
The above data is passed to d3 , and the kmeans elbow curve is plotted on the d3 side using the code snippet as given below :

```
function kmeans(error,data){
    enableSplitPlot();
    enableSinglePagePlot('#fullpage');
   kmeans_data = data["0"];
 plotlinear('#Kmeans_Elbow',kmeans_data,'Squared Error','K')
}
function plotlinear(id, data,ylabel,xlabel) {
 var margin = {
         top: 10,
         right: 10,
         bottom: 65,
```

```
    left: 180
},
width = 500 - margin.left - margin.right,
height = 300 - margin.top - margin.bottom;

 var xScale = d3.scale.linear().domain([0, data.length]).range([0, width]);
 var yScale = d3.scale.linear().domain([0, d3.max(data)]).range([height, 0]);

 // create a line function that can convert data[] into x and y points
 var line = d3.svg.line()
    .x(function(d,i) {
       return xScale(i);
    })
    .y(function(d) {
       return yScale(d);
    })

      // define x axis and y axis
var xAxis = d3.svg.axis()
    .scale(xScale)
    .orient("bottom");

var yAxis = d3.svg.axis()
    .scale(yScale)
    .orient("left");

var svg = d3.select(id).append("svg")
    .attr("width", width + margin.left + margin.right)
    .attr("height", height + margin.top + margin.bottom)
    .append("g")
    .attr("transform", "translate(" + margin.left + "," + margin.top + ")");

svg.append("g")
    .attr("class", "x axis")
    .attr("transform", "translate(0," + height + ")")
    .call(xAxis)
    .append("text")
    .attr("text-anchor", "end")
    .attr("x", width)
    .attr("y", height - 180)
    .text(xlabel)
displacement = '-3em';
if (xlabel==="K"){
displacement = '-7em';
}
svg.append("g")
    .attr("class", "y axis")
    .call(yAxis)
    .append("text")
    .attr("transform", "rotate(-90)")
    .attr("x", -height / 2)
    .attr("dy", displacement)
    .style("text-anchor", "middle")
    .text(ylabel);

 svg.append("svg:path").attr("d", line(data));

  var lineData = [{"x":0, "y": yScale(1)}, {"x": width, "y": yScale(1)}]

   var func = d3.svg.line()
```

```
            .x(function(d) {return d.x;})
            .y(function(d) {return d.y;})
            .interpolate("linear")

        var lineGraph = svg.append('path')
            .attr("d", func(lineData))
            .attr("stroke", "black")
            .attr("stroke-width", 2)
            .attr("fill", "none")
}
```

The above code plots the graph for varying values of K against the mean squared error in each case . The plot obtained is as given below :



From the above plot we can see that the optimum value for k that minimizes the error is 3. So I create 3 clusters from the data using the code snippet given below :

```
def get_clusters(data):
    cluster ={
        "0": list(),
        "1": list(),
        "2": list()
    }
    k_means = Kcluster.KMeans(n_clusters=3).fit(data).labels_
    for i in range(len(k_means)):
        cluster[str(k_means[i])].append(data[i])
    return cluster
```

The 3 clusters created from the above method is used to create the stratified sample. 30% of the data from each cluster is chosen randomly to create the stratified sample. The random sample also uses the random method from the python random package.

```
def random_sample(data):
    rsample = random.sample(data, (int)(len(data) * 0.3))
    return rsample

def stratified_sample(data):
    clusters=get_clusters(data)
    ssample =[]
    for i in clusters:
        ssample +=
random.sample(clusters[str(i)],(int)(len(clusters[str(i)])*0.3))
    return ssample
```

2. Dimension Reduction

- Intrinsic dimensionality using PCA and scree plot.

To compute the PCA , first the sample data is normalized and the correlation between the attributes are computed using the below methods.

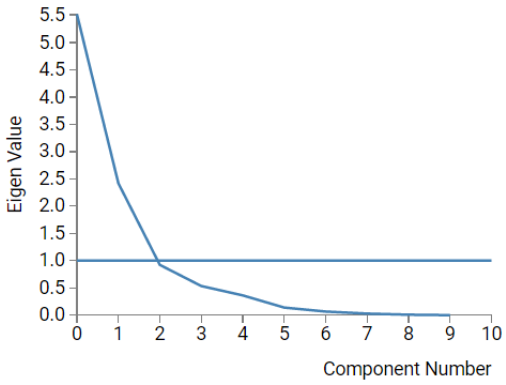
```
#Normalize the data
random_norm_data = preprocessing.scale(np.array(rsample))
stratified_norm_data = preprocessing.scale(np.array(ssample))


#Compute the correlation
random_corr_data = np.corrcoef(random_norm_data, rowvar=False)
stratified_corr_data = np.corrcoef(stratified_norm_data, rowvar=False)
```

The correlation data is then taken to compute the eigenvalues and the PCA.

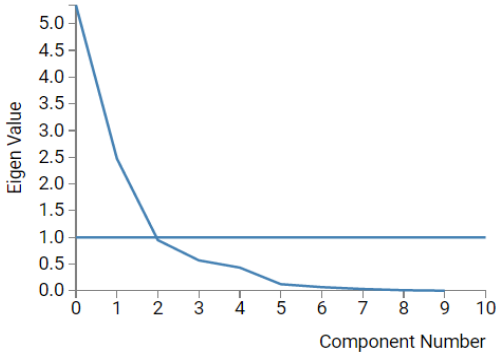
```
        def intrinsic():
            global random_corr_data
            global stratified_corr_data

            eig_randum = get_eigenvalues(random_corr_data)
            eig_stratified = get_eigenvalues(stratified_corr_data)

            combined_data_scree = {
                "random":eig_randum.tolist(),
                "stratified":eig_stratified.tolist()
            }
            combined_data_scree = json.dumps(combined_data_scree,
        default=json_util.default)
            return combined_data_scree
          def get_eigenvalues(data):
            U, V, W = np.linalg.svd(data, full_matrices=False)
            return V


        This data is then passed to the d3 layer to create the scree plot. The
        below code will create the linear plot, where the function plotlinear is
        same as the one used above.

function intrinsic(error, data) {
          enableSplitPlot();
         d3.selectAll("svg")
           .remove();
      randomdata = data["random"];
      plotlinear('#random_sample',randomdata,'Eigen Value','Component Number');
      stratifieddata = data["stratified"];
      plotlinear('#stratified_sample',stratifieddata,'Eigen Value','Component
Number' );

}
```

The scree plot obtained for random and stratified sample is shown below :

Random Sampling



Stratified Sampling



From both the plots, we can see that eigen value greater than 1 gives 3 principal components .So from the result, we get that the intrinsic dimensionality of the data is 3.

The below method is used to derive the principal components:

```
#Perform PCA on the correlation matrix. The output of the SVD phase contains the
eigenvectors and eigenvalues.
def perf_pca(data):
    U, V, W = np.linalg.svd(data, full_matrices=False)
    # print(U)
    newV = [x for x in V if x >= 1]
    # find the number of principal components.
    # select the components with eigenvalue greater than 1.
    dim = len(newV)
    sk_pca = sklearnPCA(n_components=dim)
    pca = sk_pca.fit_transform(data)
    return pca
```

- 3 attribute with highest PCA loadings.

  The principal components obtained from the above method is used to calculate the loadings for each of the attribute.

```
#Get the PCA for the correlation data.
pca_random = perf_pca(random_corr_data)
pca_stratified = perf_pca(stratified_corr_data)

# get the loadings for each PCA
loading_random = get_loadings(pca_random)
loading_stratified = get_loadings(pca_stratified)

The definition for get_loadings method is given below :

#Returns the loadings for each of the attribute from the PCA data.
def get_loadings(pcadata):
    sumofsquare = []
    for i in pcadata:
        sum = 0
        for j in i:
            sum += j*j
        sumofsquare.append(sum)
    sumofsquaredict = dict(zip(FIELD,sumofsquare))
    return sumofsquaredict
```

  The above method is used to compute the sum of squared loadings. The results of the loadings are then sorted in descending order and passed d3 code. The below code shows the sorting part.

```
def initialize():
    global loading_random
    global loading_stratified

    #Sort the data in descending order . The sorted list contains the
attribute name as well as value.
    sorted_loading_random = sorted(loading_random.items(),
key=operator.itemgetter(1),reverse=True)
    sorted_loading_stratified = sorted(loading_stratified.items(),
key=operator.itemgetter(1),reverse=True)
    combined_data_for_plot = {
        "random":sorted_loading_random,
        "stratified":sorted_loading_stratified
    }
    combined_data_for_plot = json.dumps(combined_data_for_plot,
default=json_util.default)
    return combined_data_for_plot
```

  D3 code

```
queue()
    .defer(d3.json, "/vis2/initialize")
    .await(initialize);

function initialize(error,data){
        enableSplitPlot();
        d3.selectAll("svg")
         .remove();
```

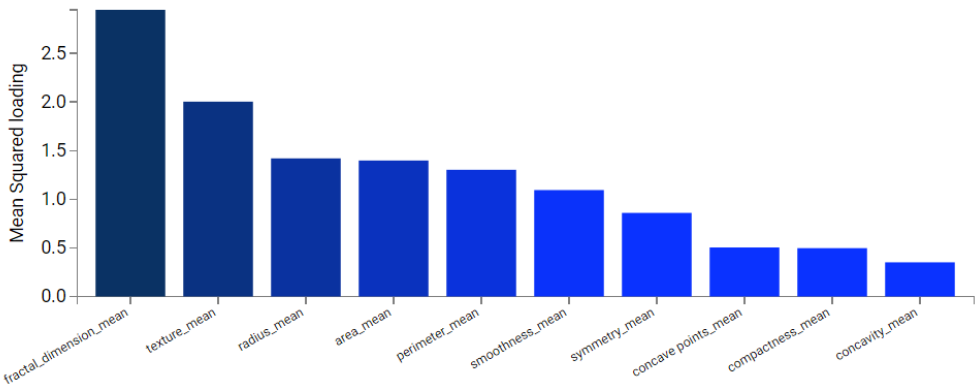
```
        var xyRandomData = [];
        //get the random data.
        randomdata = data["random"];
        for(td in randomdata){
            temp =randomdata[td];
            xyRandomData.push({
               x: temp[0],
               y: temp[1]
           })
       };
      //Render bar for the random data.
      renderBar('#random_sample',xyRandomData);

      var xyStratifiedData = [];
      //get the stratified data
        stratifieddata = data["stratified"];
        //console.log(stratifieddata);
        for(ts in stratifieddata){
            temp1=stratifieddata[ts];
            xyStratifiedData.push({
               x: temp1[0],
               y: temp1[1]
           })
       };
        //render bar for the stratified data.
      renderBar('#stratified_sample',xyStratifiedData);
}

D3 function to render the bar , given a data in the form of x,y.
function renderBar(id,barData) {
 var margin = {
          top: 10,
          right: 10,
          bottom: 68,
          left: 95,
          adjust: 20
      },
      width = 800 - margin.left - margin.right,
      height = 300 - margin.top - margin.bottom;
      var xScale = d3.scale.ordinal()
          .rangeRoundBands([0, width], 0.2, 0.2)
          .domain(barData.map(function(d) {
              return d.x;

          }));

      var yScale = d3.scale.linear()
          .range([height, 0])
          .domain([0, d3.max(barData, function(d) {
              return d.y;
          })]);

      // define x axis and y axis
      var xAxis = d3.svg.axis()
          .scale(xScale)
          .orient("bottom");

      var yAxis = d3.svg.axis()
          .scale(yScale)
          .orient("left");
```

```
    var svg = d3.select(id).append("svg")
        .attr("width", width + margin.left + margin.right)
        .attr("height", height + margin.top + margin.bottom+margin.adjust)
        .append("g")
        .attr("transform", "translate(" + margin.left + "," + margin.adjust +
")");

    svg.append("g")
        .attr("class", "x axis")
        .attr("transform", "translate(0," + height + ")")
        .call(xAxis)
        .selectAll("text")
        .attr("dx", "-.8em")
        .attr("dy", ".25em")
        .attr("transform", "rotate(-30)")
        .style("text-anchor", "end")
        .attr("font-size", "10px");

    svg.append("g")
        .attr("class", "y axis")
        .call(yAxis)
        .append("text")
        .attr("transform", "rotate(-90)")
        .attr("x", -height / 2)
        .attr("dy", "-3em")
        .style("text-anchor", "middle")
        .text("Mean Squared loading");

    var bars = svg.selectAll(".bar")
        .data( barData)
        .enter()
        .append("g")

    bars.append("rect")

        .attr("class", "rect")
        .on("mouseover", function(d, i) {
            d3.select(this)
                .attr("width", xScale.rangeBand() + 6)
                .attr("x", function(d) {
                    return xScale(d.x) - 3;
                })
                .attr("y", function(d) {
                    return yScale(d.y) - 10;
                })
                .attr("height", function(d) {
                    return height + 10 - yScale(d.y)
                })
        })
        .on("mouseout", function(d, i) {
            d3.select(this)
                .attr("width", xScale.rangeBand())
                .attr("x", function(d) {
                    return xScale(d.x);
                })
                .attr("y", function(d) {
                    return yScale(d.y);
                })
                .attr("height", function(d) {
```

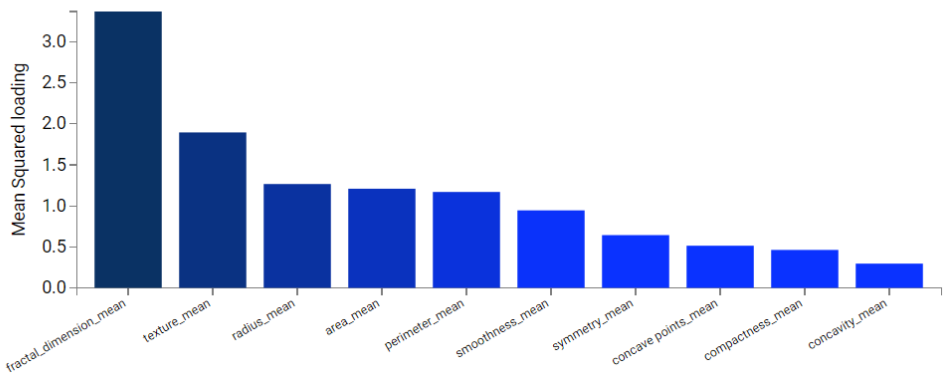
```
                    return height - yScale(d.y)
                })
            d3.select("#text" + i)
                .style("visibility", "hidden")
        })
        .attr({
            "x": function(d) {
                return xScale(d.x);
            },
            "y": function(d) {
                return yScale(d.y);
            },
            "width": xScale.rangeBand(),
            "height": function(d) {
                return height - yScale(d.y);
            }
        })

        .attr("fill", "red")

        .style("fill", function(d, i) {
            return 'rgb(10, 50, ' + ((i * 30) + 100) + ')'
        })
        .transition()
        .ease(d3.easeLinear)
        .duration(2000)
        .text(function(d) {
            return d.y;
        })
}
```

The bar graphs rendered has the sum of squared loadings on the y-axis and the attribute names on the x-axis.

Random Sampling



Stratified Sampling



From the result of the PCA loadings, the attributes with highest PCA loadings are found to be –

- Fractal_dimension_mean
- Texture_mean
- Radius_mean

It can be seen from the data that these are the attribute in the data that has the highest variance in the sample. It is logical to note that the attribute with the highest variance among the given samples is contributing more towards the reduced dimensions observed from PCA. Also, although the data had 10 attributes associated, it can be seen that only 3 (or at most 5) are significant. With the data, available for these 3 attributes alone, we will be able derive the features from the sample.

3. Visualization (Using dimension reduced data)

- Scatter plot

The data points are projected into the PCA vectors to compute the scores for each of the attribute. This is done by taking the dot product of the normalized random and stratified data. The result of the dot product is the data that is projected on the top 2 PCA dimensions. The projected data is then used to create a scatterplot.

Python code that projects the data points to the top 2 PCA vector.

```
#get the points based on PCA
points_random = np.dot(random_norm_data, pca_random)
points_stratified = np.dot(stratified_norm_data, pca_stratified)


def scatterplot():
    global points_random
    global points_stratified
```

```
    combined_data_scatter_plot = {
        "random":points_random.tolist(),
        "stratified":points_stratified.tolist()
    }

    combined_data_scatter_plot = json.dumps(combined_data_scatter_plot,
default=json_util.default)
    return combined_data_scatter_plot
```

D3 code to show the data in scatter plot:

```
function onClickScatter(e){
$('.nav').children('li').removeClass('active');
$(e.currentTarget).parent().addClass('active');
    queue()
        .defer(d3.json, "/vis2/scatterplot")
        .await(scatterplot);
}
function scatterplot(error,data){
      enableSplitPlot();
       d3.selectAll("svg")
         .remove();
      randomdata = data["random"];
      //separate out the random and stratified data from the input and call
plot_scatter to create the scatter plot for them.

    plot_scatter('#random_sample',randomdata,'Principal Component 2','Principal
Component 1');

      stratifieddata = data["stratified"];
    plot_scatter('#stratified_sample',stratifieddata,'Principal Component
2','Principal Component 1');
}
```

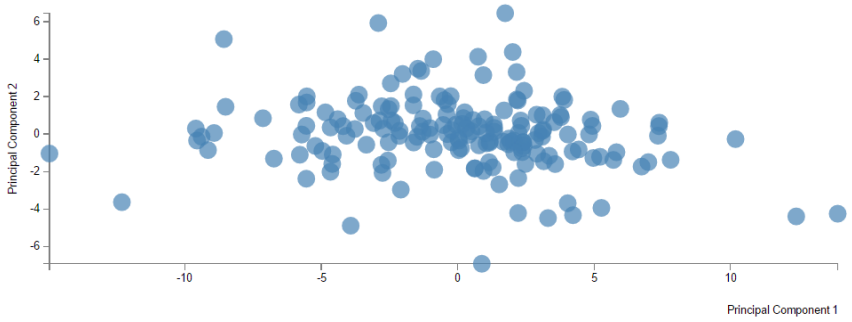
The below function creates scatter plot, given the data in 2 coordinate points:

```
function plot_scatter(id,data,ylabel,xlabel){
 var margin = {
        top: 10,
        right: 10,
        bottom: 65,
        left: 80
     },
     width = 800 - margin.left - margin.right,
     height = 300 - margin.top - margin.bottom;
     var x = d3.scale.linear()
              .domain([d3.min(data, function(d) { return d[0]; }), d3.max(data,
function(d) { return d[0]; })])
              .range([ 0, width]);

    var y = d3.scale.linear()
             .domain([d3.min(data, function(d) { return d[1]; }), d3.max(data,
function(d) { return d[1]; })])
             .range([ height, 0 ]);

    var chart = d3.select(id)
   .append('svg:svg')
```

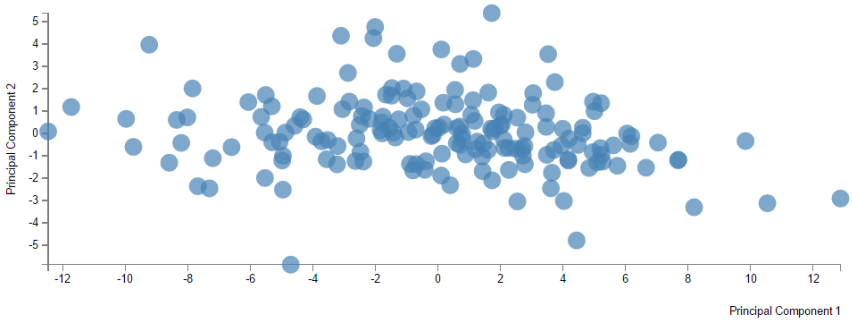
```
.attr('width', width + margin.right + margin.left)
.attr('height', height + margin.top + margin.bottom)
.attr('class', 'chart')

 var main = chart.append('g')
.attr('transform', 'translate(' + margin.left + ',' + margin.top + ')')
.attr('width', width)
.attr('height', height)
.attr('class', 'main')

 // draw the x axis
 var xAxis = d3.svg.axis()
.scale(x)
.orient('bottom');

 main.append('g')
.attr('transform', 'translate(0,' + height + ')')
.attr('class', 'main axis date')
.call(xAxis)
 .append("text")
 .attr("text-anchor", "end")
 .attr("x", width)
 .attr("y", height - 180)
 .text(xlabel)

 // draw the y axis
 var yAxis = d3.svg.axis()
.scale(y)
.orient('left');

 main.append('g')
.attr('transform', 'translate(0,0)')
.attr('class', 'main axis date')
.call(yAxis)
 .append("text")
  .attr("transform", "rotate(-90)")
  .attr("x", -height / 2)
  .attr("dy", "-3em")
  .style("text-anchor", "middle")
  .text(ylabel);

 var g = main.append("svg:g");

 g.selectAll("scatter-dots")
   .data(data)
   .enter().append("svg:circle")
       .attr("cx", function (d,i) { return x(d[0]); } )
       .attr("cy", function (d) { return y(d[1]); } )
       .attr("r", 8);}
```

The scatter plots created for both the random data and the stratified data are shown in the figures below

Random Sampling



Stratified Sampling



From the shape of the scatterplot we can see that the variance of the data is maximum among the direction of principal component 1 .The data is more scattered in the direction of principal component 1 and the mean squared error is also seen to be minimum.

- MDS Euclidean and Correlation

Euclidean MDS and Correlation MDS is computed in a comparable manner using MDS API from the sklearn manifold library. The MDS metrics is calculated using the normalized samples as input and is done for both random as well as stratified sample.

The python code to compute the Euclidean MDS is as shown below. For computing the correlation MDS we just need to pass the parameter correlation instead of Euclidean.

```
def plot_euclidean_MDS():
    global random_norm_data;
    global stratified_norm_data;
    euclidean_random_dis_mat = Met.pairwise_distances(random_norm_data, metric =
'euclidean')
    mds = MDS(n_components=2, dissimilarity='precomputed')
    euclidean_random_mds = mds.fit_transform(euclidean_random_dis_mat)

    euclidean_stratified_dis_mat = Met.pairwise_distances(stratified_norm_data,
metric = 'euclidean')
    mds = MDS(n_components=2, dissimilarity='precomputed')
    euclidean_stratified_mds = mds.fit_transform(euclidean_stratified_dis_mat)
```

```
    euclidiean_mds_for_plot = {
        "random":euclidean_random_mds.tolist(),
        "stratified":euclidean_stratified_mds.tolist()
    }
    euclidiean_mds_for_plot = json.dumps(euclidiean_mds_for_plot,
default=json_util.default)
    return euclidiean_mds_for_plot
```

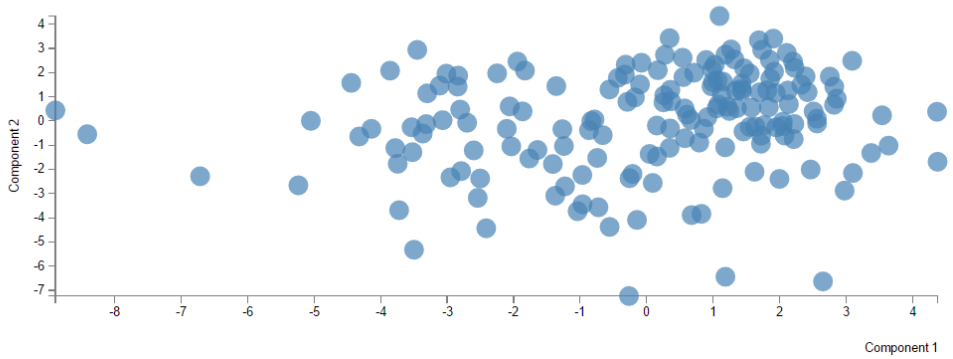
The computed MDS data from python is taken as input in the D3 side to get a scatterplot visualization. The function below shows the code required for rendering the MDS Euclidean form. Only the shape of the plots differs between MDS Euclidean and Correlation. The D3 function to visualize the MDS correlation is very similar and can be found in the source file.

```
function onClickMdsEuclidean(e){
$('.nav').children('li').removeClass('active');
$(e.currentTarget).parent().addClass('active');
    queue()
        .defer(d3.json, "/vis2/mds_euclidean")
        .await(mdsEuclidean);
}
function mdsEuclidean(error,data){
        enableSplitPlot();
        d3.selectAll("svg")
         .remove();
    randomdata = data["random"];
     plot_scatter('#random_sample',randomdata,'Component 2','Component 1');
    stratifieddata = data["stratified"];
    plot_scatter('#stratified_sample',stratifieddata,'Component 2','Component 1');
}
```

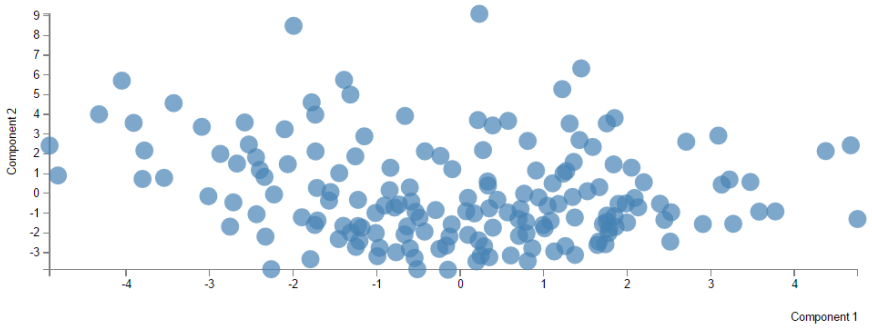
Where the plot_scatter method is same as the one shown earlier for the scatter plot.

MDS EUCLIDEAN

Random Sampling

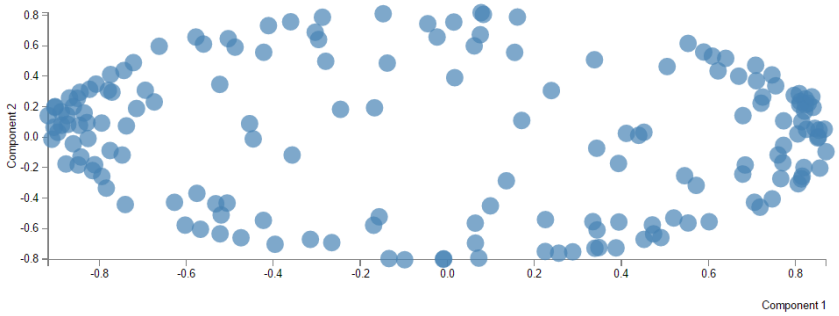


Stratified Sampling

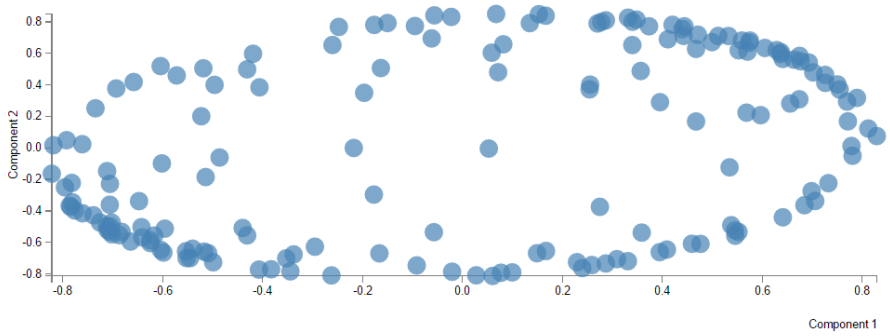


MDS CORRELATION

Random Sampling



Stratified Sampling



- Scatterplot Matrix

  The top 3 attributes as obtained from the PCA attribute loading is used to create a scatterplot matrix.

```
def plot_scatterplot_matrix():
    global loading_random
```

```
    global loading_stratified

    sorted_loading_random = sorted(loading_random.items(),
key=operator.itemgetter(1),reverse=True)

    #get the top 3 attribute from the PCA loading
    top_attr = [(i[0]) for i in sorted_loading_random[:3]]
    csvData= get_data()
    data_for_top_attr = []
    #get the data for the top 3 attribute from the datastore
    for data in csvData:
        temp =[]
        for test in top_attr:
            temp.append(data[test])
        data_for_top_attr.append(temp)
    #normalize the data obtained
    norm_data_for_top_attr = preprocessing.scale(np.array(data_for_top_attr))

    #Map the normalized values to the attribute name so as to form data of the
form - attr:value
    scatterplot_matrix_for_plot = {
        "attr":top_attr,
        "data":norm_data_for_top_attr.tolist()
    }

    scatterplot_matrix = json.dumps(scatterplot_matrix_for_plot,
default=json_util.default)
    return scatterplot_matrix
```

The data pushed to the D3 code contains the list of top 3 attributes and the sampled data corresponding to the selected attribute.

At the D3 side, the below code is used to render the scatterplot matrix.

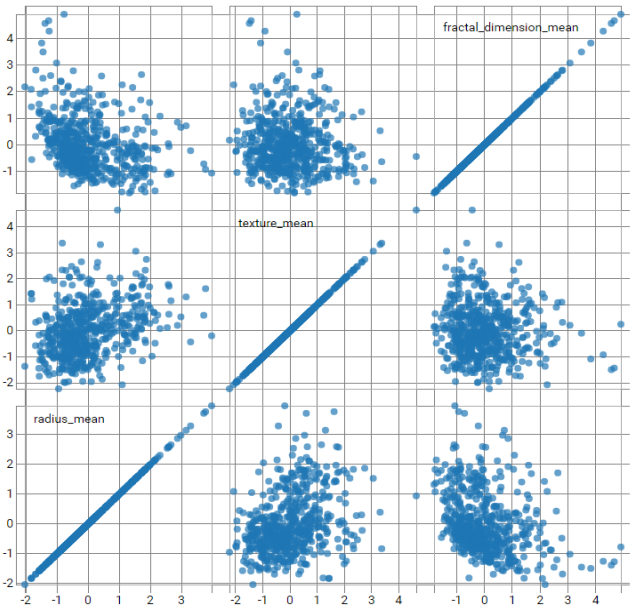
```
function onClickScatterplotMatrix(e){
$('.nav').children('li').removeClass('active');
$(e.currentTarget).parent().addClass('active');
    queue()
        .defer(d3.json, "/vis2/scatterplotmatrix")
        .await(scatterplotMatrix);
}
function scatterplotMatrix(error,input){
console.log(input);
enableSplitPlot();
enableSinglePagePlot('#fullpage2');
        d3.selectAll("svg")
         .remove();
    attr=input["attr"];
    data = input["data"];
    console.log(data);

  // Scatter plot matrix code
  var width = 960,
    size = 230,
    padding = 20;
    svgpadding=60;

var x = d3.scale.linear()
```

```
    .range([padding / 2, size - padding / 2]);

var y = d3.scale.linear()
    .range([size - padding / 2, padding / 2]);

var xAxis = d3.svg.axis()
    .scale(x)
    .orient("bottom")
    .ticks(6);

var yAxis = d3.svg.axis()
    .scale(y)
    .orient("left")
    .ticks(6);
var color = d3.scale.category10();
  var domainByTrait = {},
      traits = d3.keys(data[0]).filter(function(d) { return d}),
      n = traits.length;

  traits.forEach(function(trait) {
    domainByTrait[trait] = d3.extent(data, function(d) { return d[trait]; });
  });

  xAxis.tickSize(size * n);
  yAxis.tickSize(-size * n);
document.getElementById("matrixcard").style.height = "900px";
  var svg = d3.select("#Scatterplot").append("svg")
      .attr("width", size * n + svgpadding)
      .attr("height", size * n + svgpadding)
    .append("g")
      .attr("transform", "translate(" + svgpadding + "," + svgpadding / 2 + ")");

  svg.selectAll(".x.axis")
      .data(traits)
    .enter().append("g")
      .attr("class", "x axis")
      .attr("transform", function(d, i) { return "translate(" + (n - i - 1) * size
+ ",0)"; })
      .each(function(d) { x.domain(domainByTrait[d]); d3.select(this).call(xAxis);
});

  svg.selectAll(".y.axis")
      .data(traits)
    .enter().append("g")
      .attr("class", "y axis")
      .attr("transform", function(d, i) { return "translate(0," + i * size + ")";
})
      .each(function(d) { y.domain(domainByTrait[d]); d3.select(this).call(yAxis);
});

  var cell = svg.selectAll(".cell")
      .data(cross(traits, traits))
    .enter().append("g")
      .attr("class", "cell")
      .attr("transform", function(d) { return "translate(" + (n - d.i - 1) * size
+ "," + d.j * size + ")"; })
      .each(plot);

  // Titles for the diagonal.
  cell.filter(function(d) { return d.i === d.j; }).append("text")
```

```
      .attr("x", padding)
      .attr("y", padding)
      .attr("dy", ".71em")
      .text(function(d,i) { return attr[i]; });

  function plot(p) {
    var cell = d3.select(this);
    x.domain(domainByTrait[p.x]);
    y.domain(domainByTrait[p.y]);

    cell.append("rect")
        .attr("class", "frame")
        .attr("x", padding / 2)
        .attr("y", padding / 2)
        .attr("width", size - padding)
        .attr("height", size - padding);

    cell.selectAll("circle")
        .data(data)
      .enter().append("circle")
        .attr("cx", function(d) { return x(d[p.x]); })
        .attr("cy", function(d) { return y(d[p.y]); })
        .attr("r", 4)
        .style("fill", function(d) { return color(d.species); });
  }

function cross(a, b) {
  var c = [], n = a.length, m = b.length, i, j;
  for (i = -1; ++i < n;) for (j = -1; ++j < m;) c.push({x: a[i], i: i, y: b[j], j:
j});
  return c;
}
}
```

Below is the scatterplot matrix created using D3.The top 3 attributes are shown as labels :

ScatterPlot Matrix



Observations

- K-means forms an elbow curve. Even though the mean squared error decreases after each iteration, the clusters gets overfitted. Because if we take error alone as the measure of performance, the best cluster would happen when the number of clusters created equals the number of sample points.
- The samples created by the random and stratified forms although similar, differs slightly. The stratified sampling provides a truer representative showing more clarity in the loadings, variance and distribution among the intrinsic components. The variance is high and the squared error low for PCAs computed using the stratified sampling process showing that it gives a more representative sampling. Also as the adaptive sampling is more distributed covering outliers which are neglected by random sampling.
- The scree plot together with loading gives a very good perspective on the number of significant attributes in the data. Out of 10 components present in the original data, only 3 provides significant variation. Also, the first attribute has the most variation showing that it can describe the data well more than any other attribute.
- Although the data set has sample with large number of dimension, from the analysis we can see that the samples can be truly distinguished with using just 3 attributes from the data.
- We can see that the loadings of the attribute have an exponential decrease. This points out that out of 10 different attributes , one attribute(the one with the highest loading) provides more features to distinguish the data.
- It can be seen from the MDS plots that the distance metric/variance is maintained for the actual data as well as the in the new data.

Alternatives

- We can use other alternative clustering algorithms like DBSCAN to perform clustering for doing stratified sampling.
- Instead of random or adaptive sampling we can use representative or probabilistic sampling to get a better representation.
- We use a bar chart to better visualize the variance or mean squared error that each component of PCA is giving. This will, like scree plot, give us a knowledge of how strong each component is.
- We can use cosine distance instead of Euclidean or correlation to compute MDS . Another option would be to use the manhattan distance.