CS3481 Homework 3

In this homework, I am going to provide a report on constructing various clustering algorithms on **Vertebral Column** dataset. As discussed earlier in previous homework assignments, we have a dataset consisting with 6 feature columns and 3 possible category values: **disk hernia (DH), spondylolisthesis (SL) or Normal (NO).**

For this assignment, I am going to drop the category column from the original dataset and test whether clustering algorithms may achieve to group the data points properly as it should be by default.

```
df = pd.read_excel('./data.xlsx')
original_df = df
df = df.drop(['category'],axis = 1) #drop category column for clustering purpose
original_df.head()
```

Out[1]:

	c_incidence	c_tilt	lumbar_lordosis_angle	sacral_slope	pelvic_radius	degree_spondylolisthesis	category
0	63.03	22.55	39.61	40.48	98.67	-0.25	DH
1	39.06	10.06	25.02	29.00	114.41	4.56	DH
2	68.83	22.22	50.09	46.61	105.99	-3.53	DH
3	69.30	24.65	44.31	44.64	101.87	11.21	DH
4	49.71	9.65	28.32	40.06	108.17	7.92	DH

In [2]:
```
df.head()
```

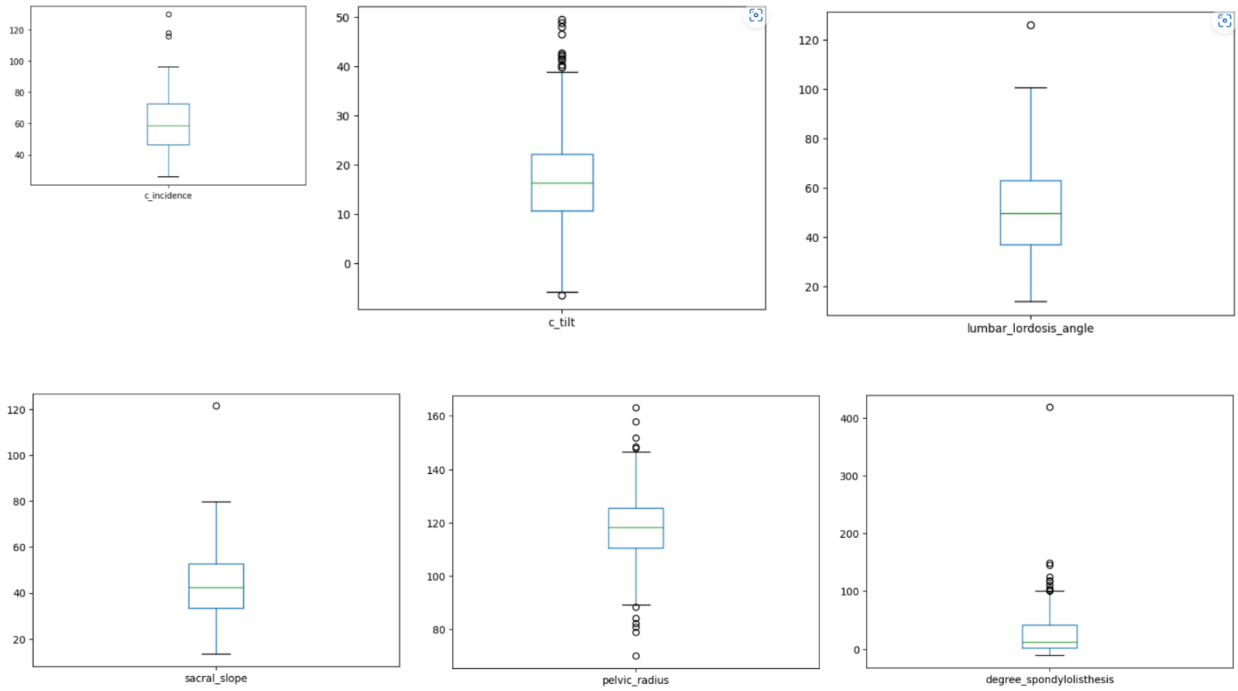
Out[2]:

	c_incidence	c_tilt	lumbar_lordosis_angle	sacral_slope	pelvic_radius	degree_spondylolisthesis
0	63.03	22.55	39.61	40.48	98.67	-0.25
1	39.06	10.06	25.02	29.00	114.41	4.56
2	68.83	22.22	50.09	46.61	105.99	-3.53
3	69.30	24.65	44.31	44.64	101.87	11.21
4	49.71	9.65	28.32	40.06	108.17	7.92

Preprocessing the dataset

First, for clustering algorithms to operate properly, the dataset has to be thoroughly preprocessed. Using a helper function 'display_boxplots()', I have displayed the boxplot diagrams for each feature column:

```
def display_boxplots(df, col):
    df.boxplot(column=[col])
    plt.grid(False)
    plt.show()
```



Here, we can clearly see that each feature column contains outliers, and we have to remove them. This is because outliers can have a significant impact on the clustering results. Outliers are data points that are significantly different from the other data points in the dataset, and they can have a disproportionate influence on the calculation of the distances between points.

In unsupervised machine learning, clustering algorithms group similar data points together based on some measure of distance or similarity. If outliers are included in the dataset, they can affect the calculation of these distances and distort the clustering results. Outliers can also form their own clusters or disrupt the formation of other clusters, leading to suboptimal clustering solutions.

By removing outliers, the clustering algorithm can focus on grouping the more representative data points together and provide a more accurate representation of the underlying structure of the data. However, it is important to note that the decision of whether or not to remove outliers should be based on domain knowledge and context-specific considerations. In some cases, outliers may be important or informative data points that should not be discarded.

I have used IQR method for detecting outliers in the dataset: if the datapoint is 1.5*Interquartile range less than Q1 quartile or 1.5*Interquartile range bigger than Q3 quartile, then it is an outlier:

```
: #As we can see, each feature column has at least 1 outlier data points. We have to remove the outliers:

  def outliers(df, col):

      Q1 = df[col].quantile(0.25)
      Q3 = df[col].quantile(0.75)
      IQR = Q3-Q1

      lower = Q1-1.5*IQR
      upper = Q3+1.5*IQR

      mean = df[col].mean()
      std = df[col].std()



      return df.index[(df[col] < lower) | (df[col] > upper)]
```

```
outlier_indexes = []

for col in df.columns:
    outlier_indexes.extend(outliers(df,col))
outlier_indexes
```

```
]: def remove_outliers(df,outliers):
       df = df.drop(list(set(outliers)))

       return df
```

```
]: df_temp = remove_outliers(df, outlier_indexes)
```

Here, I have successfully removed the outliers from the dataset. As a note, initially, the size of my original dataset was (310,6), now that outliers are removed, it is (279,6):

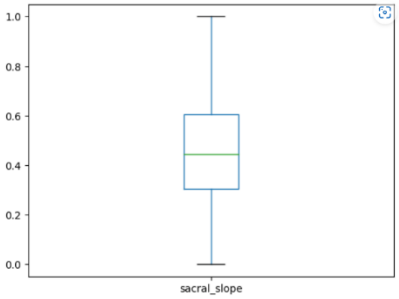
```
df.shape #before preprocessing
```

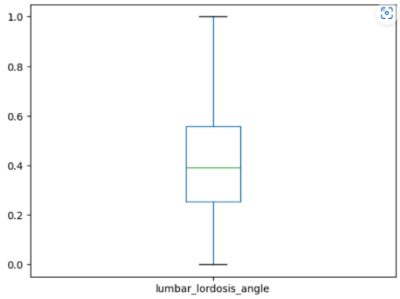
```
(310, 6)
```

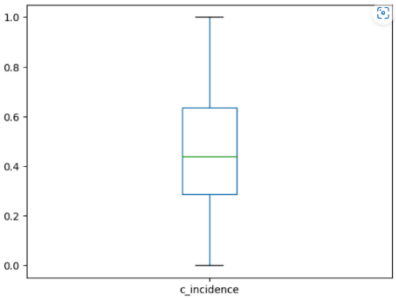
```
df_temp.shape
```

```
(279, 6)
```

Boxplots of feature columns after the removal of outliers:







We can also notice that some feature columns have differing spreads:

```
df_temp.describe()
```

	c_incidence	c_tilt	lumbar_lordosis_angle	sacral_slope	pelvic_radius	degree_spondylolisthesis
count	279.000000	279.000000	279.000000	279.000000	279.000000	279.000000
mean	59.024659	16.262652	50.801649	42.762079	117.908602	20.831792
std	15.824825	8.183525	18.169694	12.547911	11.260339	25.142615
min	26.150000	-5.850000	14.000000	13.370000	89.310000	-11.060000
25%	46.380000	10.600000	36.140000	33.355000	111.175000	1.155000
50%	56.990000	15.840000	47.870000	42.450000	118.360000	7.950000
75%	70.815000	20.980000	62.350000	52.880000	125.205000	36.915000
max	96.660000	38.750000	100.740000	78.790000	145.600000	100.290000

Based on the corresponding mean and std values, we can conclude that the dataset has to be scaled into the same standard. Handling different scaling measures of feature columns is important in clustering because clustering algorithms are sensitive to the scale of the data. If the features have different scales, then features with larger scales will have a greater influence on the distance or similarity measures used by the clustering algorithm. As a result, the clustering solution may be dominated by those features with larger scales, while other features with smaller scales may be overlooked.

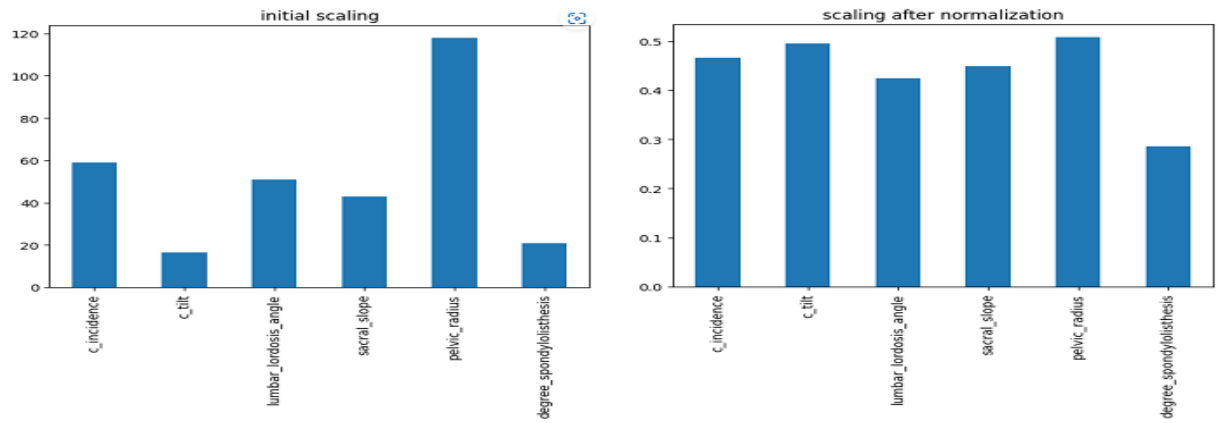
Normalization is a common way to handle different scaling measures of feature columns in clustering. Normalization scales the values of each feature to a common scale, typically between 0 and 1 or -1 and 1. This ensures that each feature contributes equally to the clustering algorithm, regardless of its original scale. By normalizing the data, the clustering algorithm can focus on the relative differences between the features rather than their absolute values.

There are various methods of normalization that can be used, such as min-max scaling, z-score normalization, and unit vector scaling. Since the dataset has already been cleaned from outliers, we can use min-max scaling for this assignment.

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()

df = pd.DataFrame(scaler.fit_transform(df_temp), columns=df_temp.columns)

df.describe()
```

	c_incidence	c_tilt	lumbar_lordosis_angle	sacral_slope	pelvic_radius	degree_spondylolisthesis
count	279.000000	279.000000	279.000000	279.000000	279.000000	279.000000
mean	0.466241	0.495799	0.424275	0.449283	0.508058	0.286410
std	0.224434	0.183487	0.209473	0.191805	0.200042	0.225798
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.286910	0.368834	0.255246	0.305488	0.388435	0.109699
50%	0.437385	0.486323	0.390477	0.444512	0.516077	0.170723
75%	0.633456	0.601570	0.557413	0.603944	0.637680	0.430849
max	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000

Now that the dataset has been normalized, we can compare the scaling before and after the normalization:



1) *Compare the hierarchical structures generated using single link, complete link and group average for the Vertebral Column data set. (30%)*

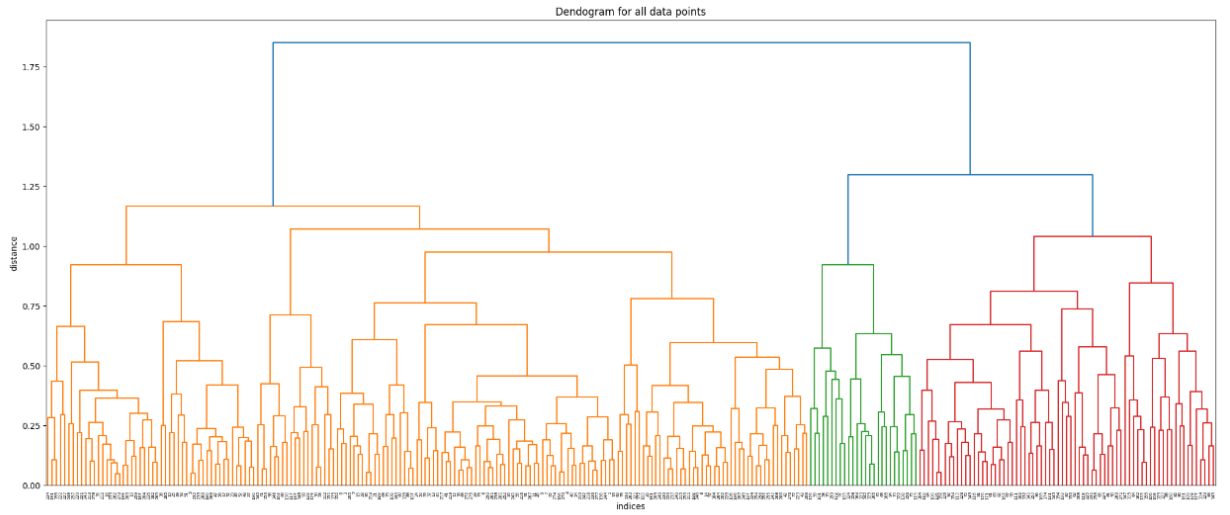
I have initialized 3 linkage matrices for corresponding 3 linking variations:

```
# now that the dataset is preprocessed, we may start applying Clustering Algorithms for the purpose of Assignment3.

from sklearn.cluster import AgglomerativeClustering

features = ['c_incidence','c_tilt','lumbar_lordosis_angle','sacral_slope','pelvic_radius','degree_spondylolisthesis']
X = df[features]

Z_complete = linkage(X,'complete')
Z_single = linkage(X,'single')
Z_group = linkage(X,'average')
```

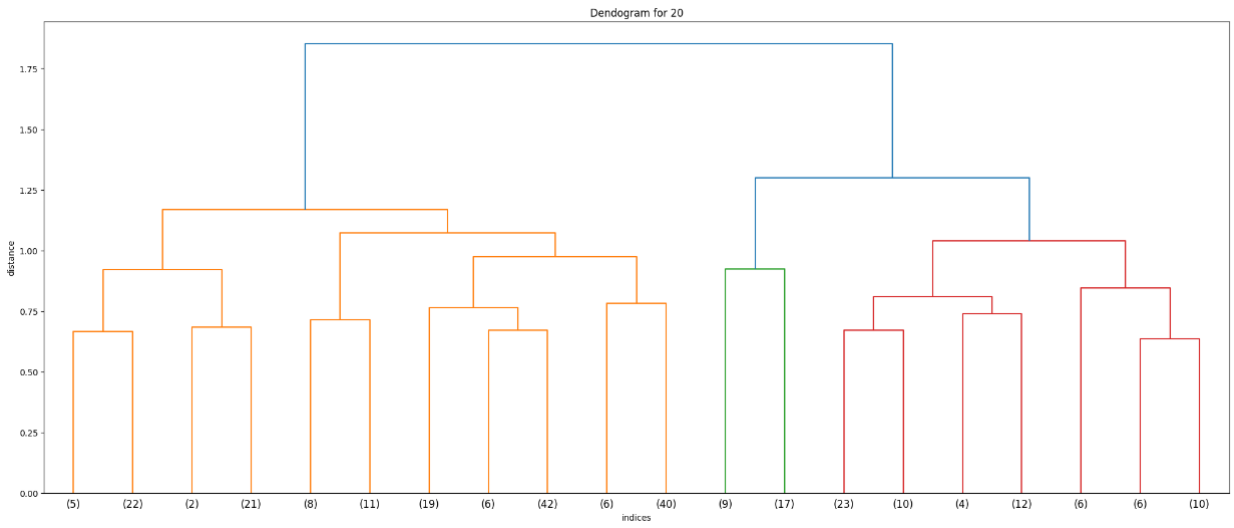
Here is the dendrogram for Hierarchical Clustering with Complete Link:

For hierarchical structure generated using Complete Link

```
display_dendrogram(Z_complete)
```



As there are 279 singleton clusters in the beginning, the dendrogram may seem a bit complex and complicated. Here is the dendrogram built for 20 starting singleton clusters:

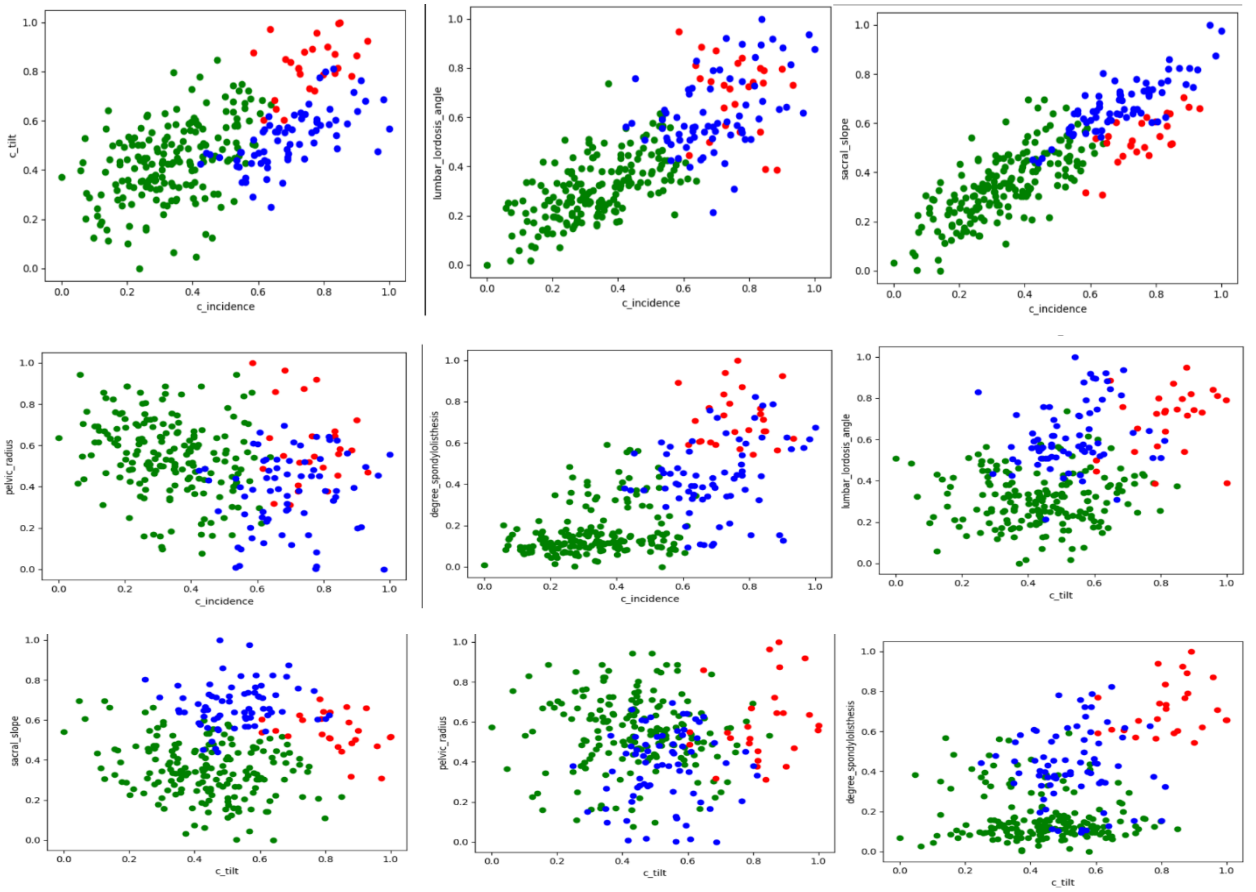


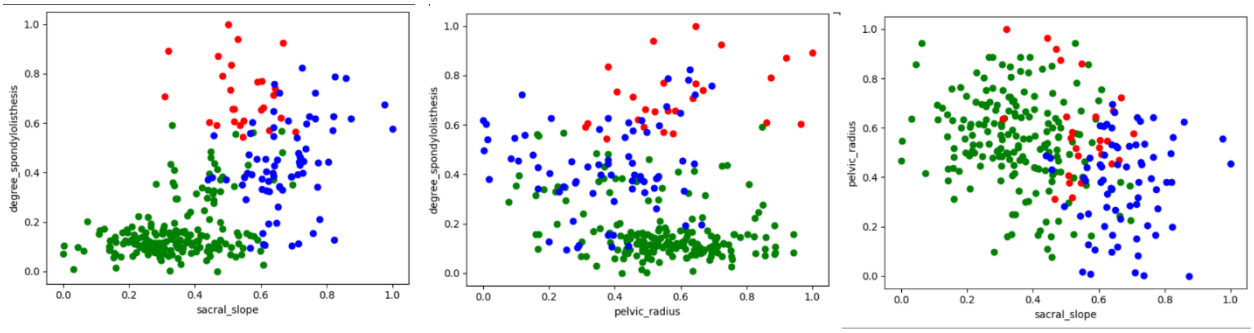
Although it might not seem crystal clear, but we may favor choosing the number of clusters as 3.

Since we have 6 feature columns, it is both computationally and intuitively difficult to plot it using scatterplot. Hence, I have decided to plot the relationship between each 2 feature columns only:

```
def display_clusters_2d(df,ft1,ft2):
    df1 = df[df.cluster==1]
    df2 = df[df.cluster==2]
    df3 = df[df.cluster==3]

    plt.scatter(df1[ft1],df1[ft2],color='green')
    plt.scatter(df2[ft1],df2[ft2],color='red')
    plt.scatter(df3[ft1],df3[ft2],color='blue')

    plt.xlabel(ft1)
    plt.ylabel(ft2)
    plt.show()
```

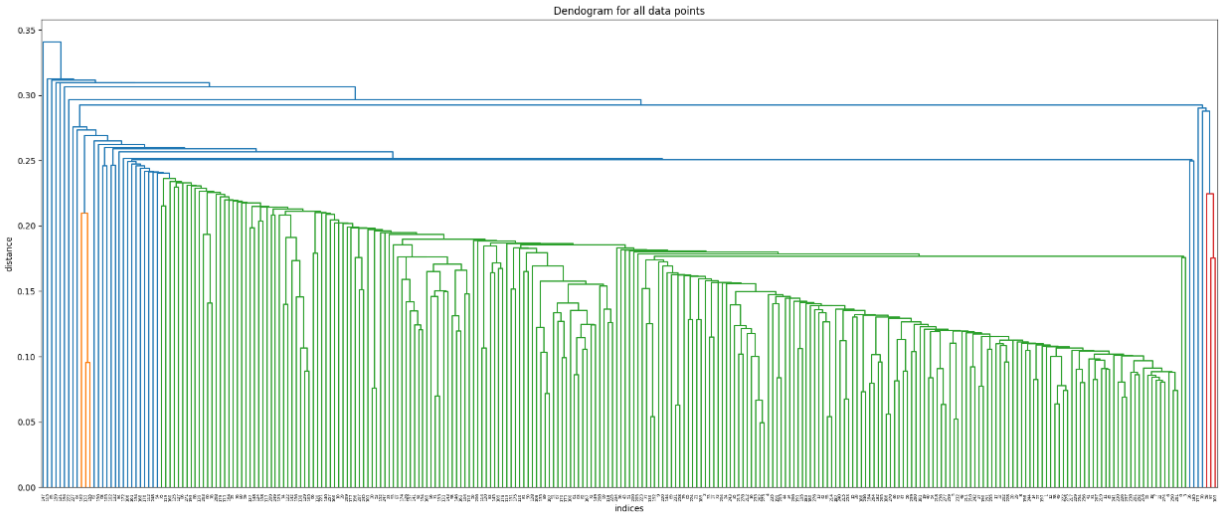
```
def display_clusters(df):
    s = set()
    for col1 in df.columns:
        for col2 in df.columns:
            if col1!=col2 and col1!='cluster' and col2!='cluster':
                if (col1,col2) not in s and (col2,col1) not in s:
                    s.add((col1,col2))
                    s.add((col2,col1))
                    display_clusters_2d(df,col1,col2)
```



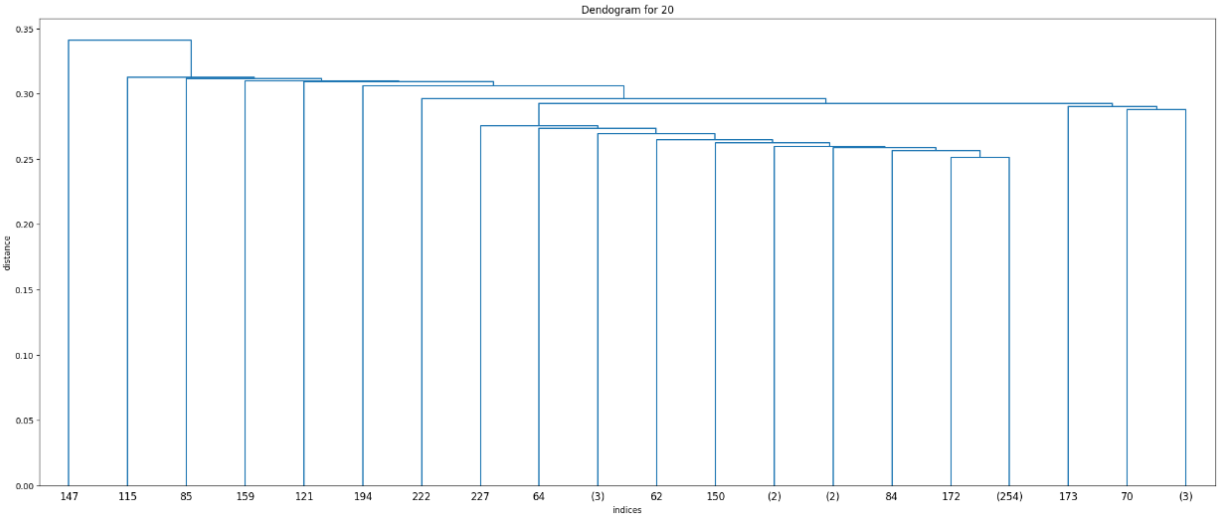


Although this method might seem inconclusive to some extent, I deemed it right to use for the purpose of comparison when using different types of linkage. Hence, for Single Link the dendrogram looks like this:

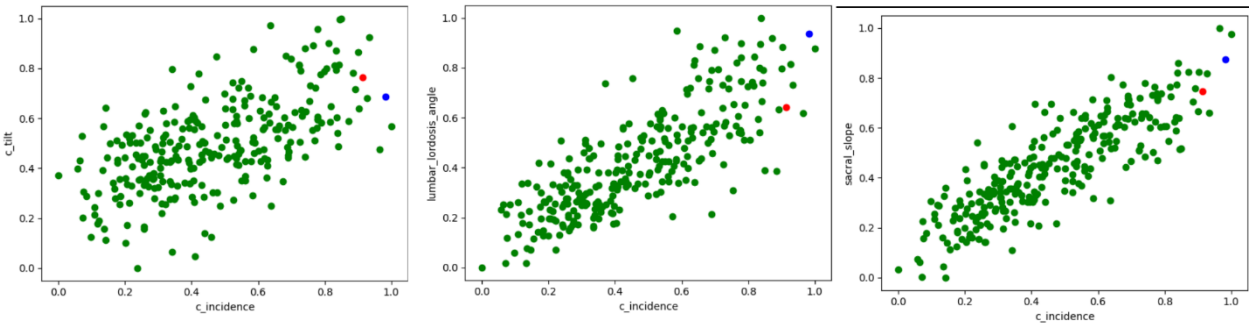
```
display_dendrogram(Z_single)
```




```
display_dendrogram_for_20(Z_single)
```



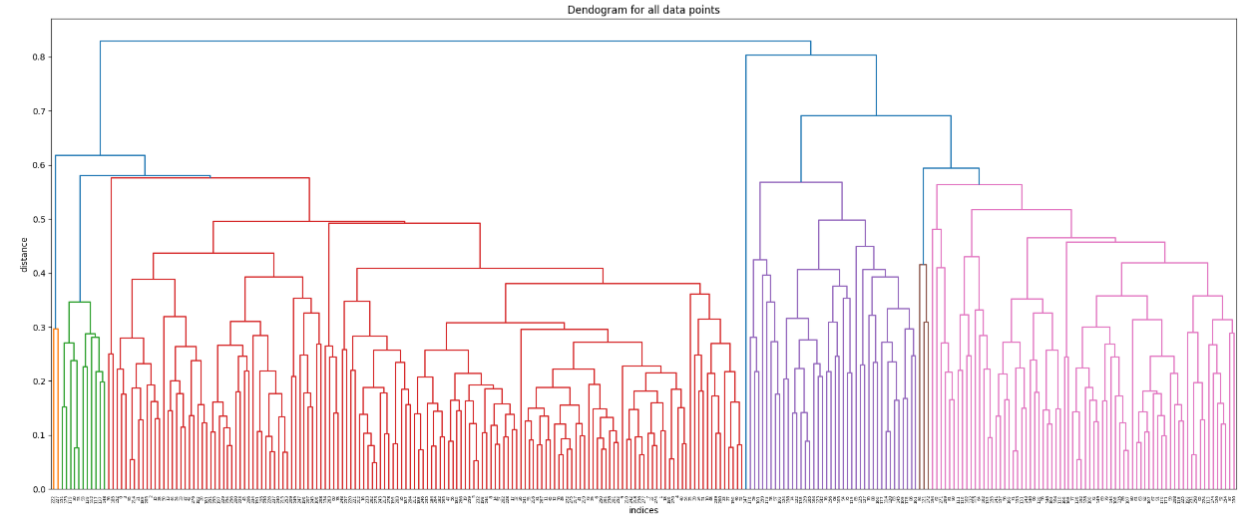
We can clearly tell from both dendrogram and scatterplots that single link usage **will not** help finding the best clustering:

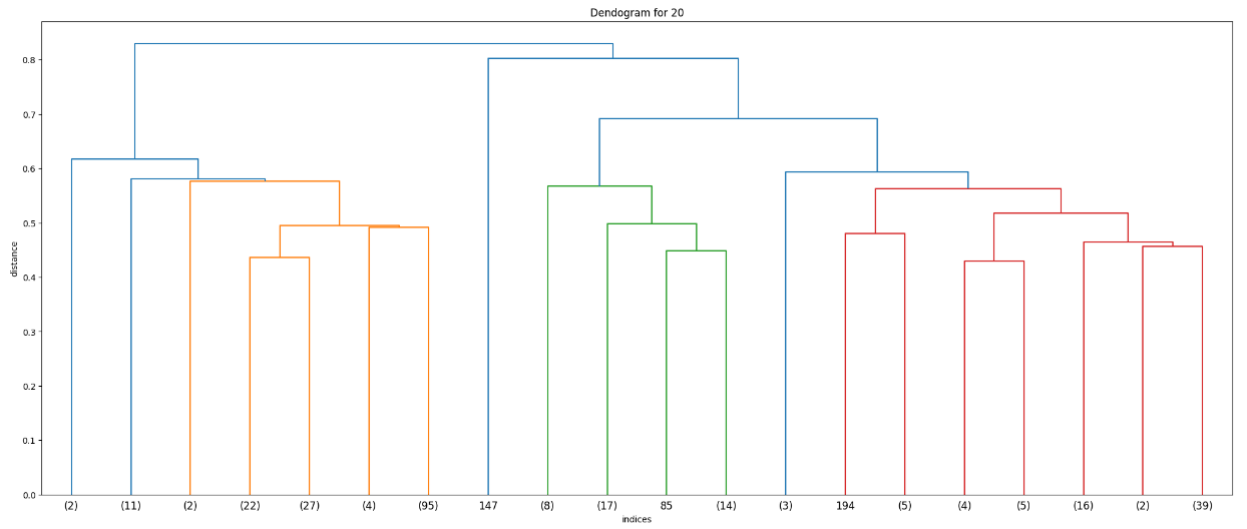


And finally, when it comes to the Group Average Link usage, the dendrogram seems less chaotic hence more conclusive than that of Single Link:

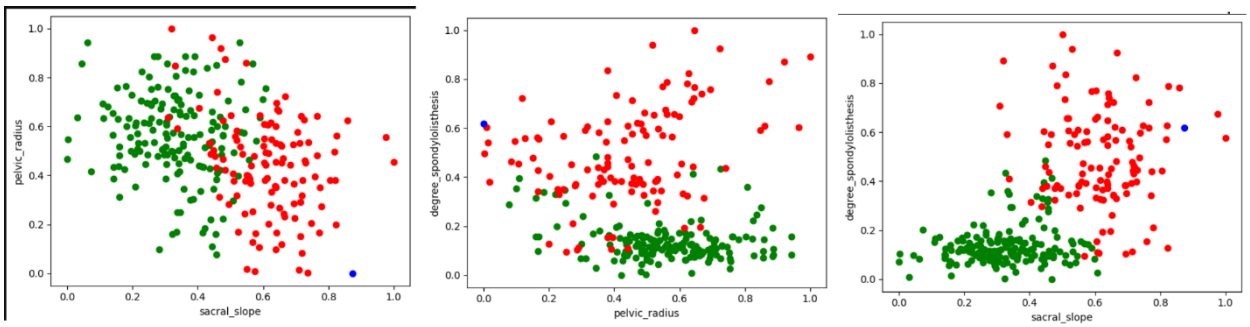
For hierarchical structure generated using Group Average Link

```
display_dendrogram(Z_group)
```





However, based on the scatterplots, it mostly suggests clustering into 2 groups which may not converge with the number of groups in the original number of categories:



2. *For some of these hierarchical structures, observe the set of distance values at which cluster merge occurs, and identify possible patterns from these values. (20%)*

Recalling the definitions of different types of linkages:

- Complete Link: The distance between two clusters is defined as the maximum distance between any pair of points in the two clusters. This algorithm tends to produce compact, spherical clusters and is sensitive to outliers.
- Single Link: The distance between two clusters is defined as the minimum distance between any pair of points in the two clusters. This algorithm tends to produce long, thin clusters and is sensitive to noise.
- Group Average Link: The distance between two clusters is defined as the average distance between all pairs of points in the two clusters. This algorithm is less sensitive to outliers and noise than complete and single link and tends to produce more balanced clusters.

The 'linkage()' function returns a matrix with *2 clusters indices, distance between clusters and the size of the resulting cluster.*

Hence, we may use the resulting linkage matrix for the purpose of this part of the assignment:

```
def display_distance_vals(df, z):

    distances = Z_complete[:, 2]

    # Record the distance values in a table
    table = []
    for i in range(len(distances)):
        cluster1 = int(Z_complete[i, 0])
        cluster2 = int(Z_complete[i, 1])
        size = int(Z_complete[i, 3])
        table.append([distances[i], cluster1, cluster2, size])
    table.reverse()
    # Print the table
    print("Distance\tCluster 1\tCluster 2\tSize")
    for row in table:
        print("{:.3f}\t\t{}\t\t{}\t\t{}".format(row[0], row[1], row[2], row[3]))
```

For Z_complete, the function prints out the distance, cluster1 and cluster2 and the size:

0.080	254	256	2
0.080	184	264	2
0.077	27	190	2
0.076	30	53	2
0.074	250	261	2
0.074	272	275	2
0.073	188	246	2
0.072	89	107	2
0.070	71	153	2
0.069	226	237	2
0.068	215	253	2
0.064	38	49	2
0.063	236	277	2
0.063	231	258	2
0.056	186	270	2
0.054	95	214	2
0.054	136	192	2
0.052	5	232	2
0.049	262	273	2

display_distance_vals(df,Z_complete)

Distance	Cluster 1	Cluster 2	Size
1.851	554	555	279
1.299	550	552	97
1.167	549	553	182
1.072	543	551	132
1.040	547	548	71
0.975	545	546	113
0.922	533	537	26
0.922	539	542	50
0.846	529	538	22
0.810	541	544	49
0.782	523	535	46
0.763	536	540	67
0.738	514	534	16
0.713	511	522	19
0.684	445	525	23
0.672	526	531	33
0.672	491	518	48
0.665	513	524	27

Z_single linkage matrix:

display_distance_vals(df,Z_single)

0.078	286	293	4
0.077	27	190	2
0.076	30	53	2
0.074	250	261	2
0.074	272	275	2
0.073	188	246	2
0.072	89	107	2
0.070	71	153	2
0.069	226	237	2
0.068	215	253	2
0.066	233	279	3
0.064	38	49	2
0.063	236	277	2
0.063	231	258	2
0.056	186	270	2
0.054	95	214	2
0.054	136	192	2
0.052	5	232	2
0.049	262	273	2

display_distance_vals(df,Z_single)

Distance	Cluster 1	Cluster 2	Size
0.341	147	555	279
0.312	115	554	278
0.311	85	553	277
0.310	159	552	276
0.310	121	551	275
0.306	194	550	274
0.297	222	549	273
0.292	546	548	272
0.290	173	547	5
0.288	70	515	4
0.275	227	545	267
0.274	64	544	266
0.269	499	543	265
0.265	62	542	262
0.262	150	541	261
0.260	532	540	260
0.259	533	539	258
0.257	84	538	256

Z_group linkage matrix:

display_distance_vals(df,Z_group)

0.080	254	256	2
0.080	184	264	2
0.077	27	190	2
0.076	30	53	2
0.074	250	261	2
0.074	272	275	2
0.073	188	246	2
0.072	89	107	2
0.070	71	153	2
0.069	226	237	2
0.068	215	253	2
0.064	38	49	2
0.063	236	277	2
0.063	231	258	2
0.056	186	270	2
0.054	95	214	2
0.054	136	192	2
0.052	5	232	2
0.049	262	273	2

display_distance_vals(df,Z_group)

Distance	Cluster 1	Cluster 2	Size
0.829	553	555	279
0.803	147	554	116
0.691	549	552	115
0.618	499	551	163
0.594	535	548	75
0.581	517	550	161
0.576	469	545	150
0.568	537	546	40
0.563	543	547	72
0.517	538	542	66
0.498	531	540	32
0.495	539	544	148
0.492	475	533	99
0.480	194	534	6
0.465	519	541	57
0.457	459	536	41
0.449	85	532	15
0.436	527	528	49

As all data points are initialized as singleton clusters, the merge occurs identically at the beginning for every linkage type. This is because the cluster sizes are only 1 hence the max distance, min distance and grouped average distance are totally the same. As size of clusters gets bigger, they will start having bigger spreads. Therefore, for complete linkage algorithm, it will start favoring the distance between the points located farthest from each cluster respectively. For instance, when Cluster with index 554 and index 555 are to be merged, the resulting size is 279 which is huge: the algorithm will favor the points located on the far edges of opposite sides of both clusters, thus the distance is 1.851 which is the farthest.

However, for single linkage algorithm, it will do a completely opposite thing: when the size of merged cluster is 279 which is huge, because of the higher spread in the cluster, the algorithm can find closest points from 2 previous clusters, thus the distance is 0.341 which is the shortest.

Alternatively, for groupaverage linkage algorithm, the distance between two clusters is defined as the average distance between all pairs of points in the two clusters. Therefore, the distance at which the final cluster with size 279 is to be merged is 0.829 which is not the farthest (1.851) nor shortest (0.341).

3. *Select different clustering solutions from the hierarchical structures, and compare the cluster groupings with the corresponding K-means clustering solutions (using the method KMeans from the module sklearn.cluster), in terms of the extent to which the clusters can capture the class structure of the data set. (30%)*

As discussed earlier in part 1, the usage of single link or group average link shows vague clustering results. Hence, I will be using complete linkage from now on in order to compare its performance to that of Kmeans clustering algorithm.

Frankly, initially, I was going to apply Kmeans to a data without removal of outliers in order to show how the algorithm performs poorly because of outliers. This is because K-means is sensitive to outliers because it tries to minimize the sum of squared distances between the data points and the centroid of their assigned cluster. Outliers can greatly impact this objective function by increasing the sum of squared distances, which can result in the centroids being pulled away from the dense clusters towards the outliers. **However**, the result is surprising, and I will be discussing about that shortly after.

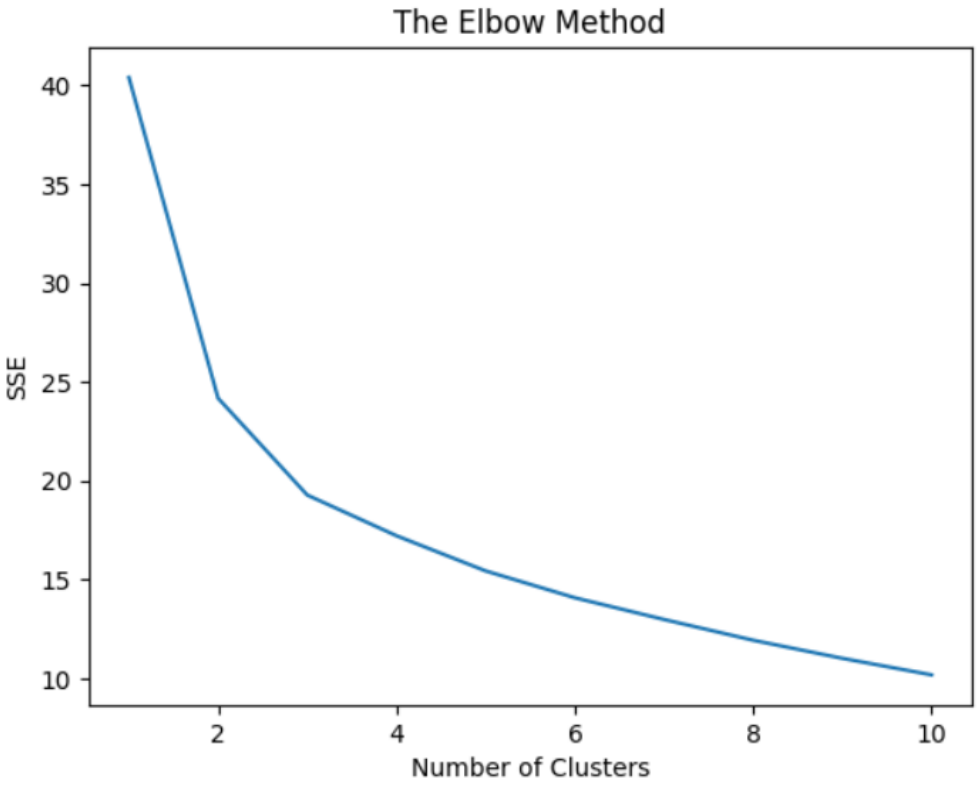
First, in order choose the most optimal number of clusters, I have applied elbow method. The elbow method is a heuristic technique used to determine the optimal number of clusters in a dataset. It involves plotting the sum of squares errors (SSE) against the number of clusters and selecting the number of clusters at the "elbow" of the plot. The elbow point is the point of inflection on the curve, where the rate of decrease of SSE slows down and starts to level off.

The idea behind the elbow method is that as the number of clusters increases, the SSE should decrease since the clusters become more and more similar to their respective centroids. However, beyond a certain point, adding more clusters does not significantly reduce the SSE, as the clusters become too small and overfit the data. The elbow method helps to identify the point at which the SSE reduction slows down, indicating that adding more clusters does not provide significant improvement in clustering quality.

Therefore, the optimal number of clusters is typically chosen at the elbow point of the curve, as it represents the trade-off between maximizing the clustering quality and minimizing the complexity of the model.

The following code generates the graph for elbow method:

```
feature_kmeans = df_kmeans.columns
X_kmeans = df_kmeans[feature_kmeans]
sse=[]
for i in range(1,11):
    kmeans=KMeans(n_clusters=i, init='k-means++',random_state=0)
    kmeans.fit(X_kmeans)
    sse.append(kmeans.inertia_)

plt.plot(range(1,11),sse)
plt.title('The Elbow Method')
plt.xlabel('Number of Clusters')
plt.ylabel('SSE')
plt.show()
```



By the graph, we can see that the 'elbow' is around n_clusters = 3. Hence, we choose n_clusters as 3 from now on.

```
kmeans=KMeans(n_clusters=3)
kmeans.fit(X_kmeans)
kmeans.labels_ = kmeans.labels_+1 #make 1,2,3
kmeans.labels_
```

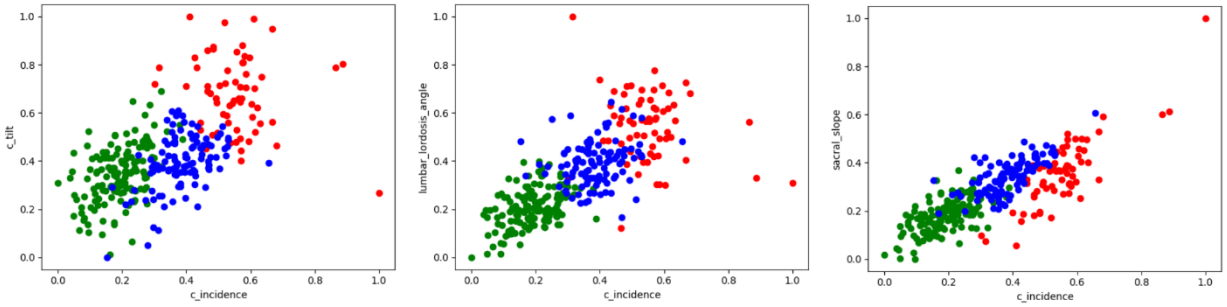
```
X_kmeans['cluster'] = kmeans.labels_ #add 'cluster' column with respective values

kmeans_counts = X_kmeans['cluster'].value_counts()

print(f"Original dataset category proportions : {original_df['category'].value_counts()}")
print(f"Kmeans clusters proportions : {kmeans_counts}")
print(f"Hierarchical clusters proportions after preprocessing : {df['cluster'].value_counts()}")
```

The following code generated a surprising result:

```
Original dataset category proportions : SL    150
NO    100
DH     60
Name: category, dtype: int64
Kmeans clusters proportions : 1    140
3    105
2     65
Name: cluster, dtype: int64
Hierarchical clusters proportions after preprocessing : 1    163
2    115
3      1
Name: cluster, dtype: int64
```

It is clearly seen that Kmeans clusters performs almost perfect for this dataset, given the fact the initial category split was 150/100/60, the Kmeans algorithm has formed 3 groups with 140/105/65 split. This is in fact surprising in a sense that I have not applied outlier removal technique for the dataset for Kmeans algorithm. Even with outliers in the dataset, the algorithm was able to perform much better than the hierarchical cluster with complete link.

Here is an example of how Kmeans clusters the datapoints with respect to certain feature columns:



Given the fact that Kmeans algorithm performed almost close to ideal, we can encode the categories of our original dataset by matching it to an outcome of Kmeans like this: SL -> 1, NO -> 3, DH -> 2.

```
for_map = {'SL':2,'NO':1,'DH':3}
original_df['category']=original_df['category'].map(for_map)
original_df['category'].value_counts()
```

```
2    150
1    100
3     60
Name: category, dtype: int64
```

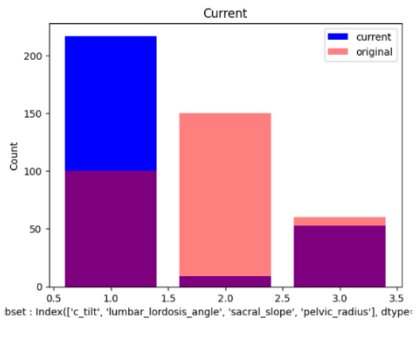
4. *Select different subsets of attributes from the data sets and re-perform hierarchical clustering. Compare the resulting hierarchical structures based on the selected attribute subsets with the original hierarchical structures. (20%)*

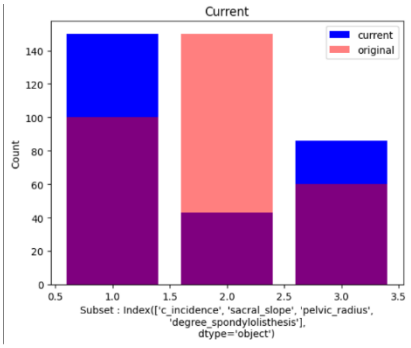
For part (3), we have used class proportion comparison in order to evaluate the result of clustering algorithms.

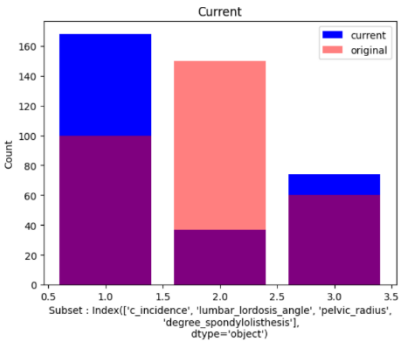
For this part, main motivation is to generate cluster proportions (ratio) for each **proper** subset of features to see if certain combinations match the original class proportion. To be more specific, if we have a feature F={A,B,C,D}, the goal is to build a clustering algorithm for {A,B,C},{A,B,D},{B,C,D},{A,C,D},{A,B},{A,C}, etc., and see the value_counts().

```
import itertools

temp_df = df.copy()
original_counts = original_df['category'].value_counts()


for r in range(2,len(df.columns)):
    for subset in itertools.combinations(df.columns,r):
        subset_list = df[list(subset)]
        linkage_matrix = linkage(subset_list, method='complete')
        add_cluster_column(temp_df,linkage_matrix)
        counts = temp_df['cluster'].value_counts()
        fig, ax = plt.subplots()

        ax.bar(counts.index,counts.values,color='blue',label='current')
        ax.bar(original_counts.index, original_counts.values, color='red',label='original',alpha=0.5)
        ax.set_xlabel(f'Subset : {subset_list.columns}')
        ax.set_ylabel('Count')
        ax.set_title('Current')
        ax.legend()



        plt.show()
```

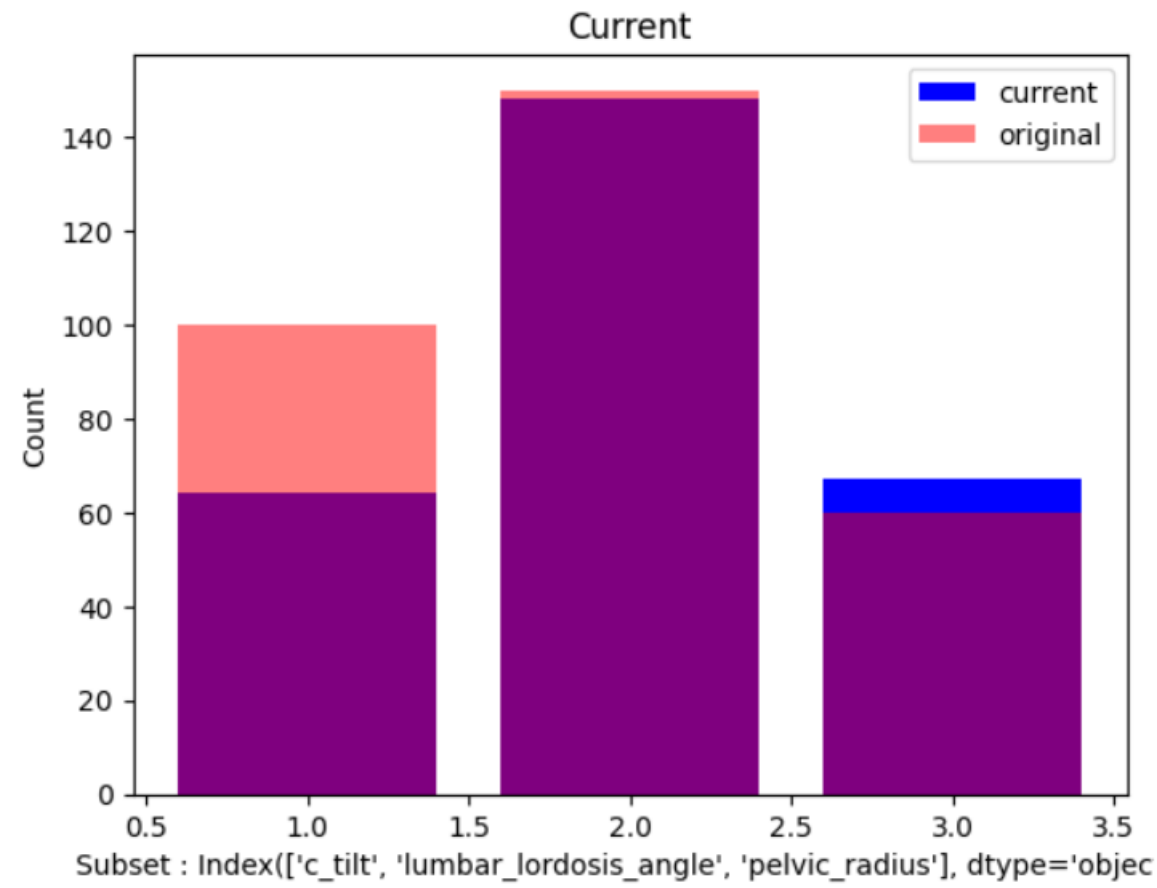
This code implements the idea and also generates a barplot of original dataset's proportions – 150,60,100 – along with the barplot of current subset's clustering results. As the number of features is 6, there are 2**6-6-1=57 possible subsets that can be formed with various results:







From the generated barplots, we can select those whose values converge to those of original dataset. For instance, if we choose ['c_tilt', 'lumbar_lordosis_angle','pelvic_radius'], it can generate similar clustering splits:



I have manually selected subsets that are closer to intended grouping and printed out their grouping results as shown below:

```
needed_subsets = [['c_incidence','sacral_slope'],['c_incidence','c_tilt','sacral_slope'],['c_tilt','lumbar_lordosis_angle','pelvi
                  ['sacral_slope','pelvic_radius','degree_spondylolisthesis']]
```

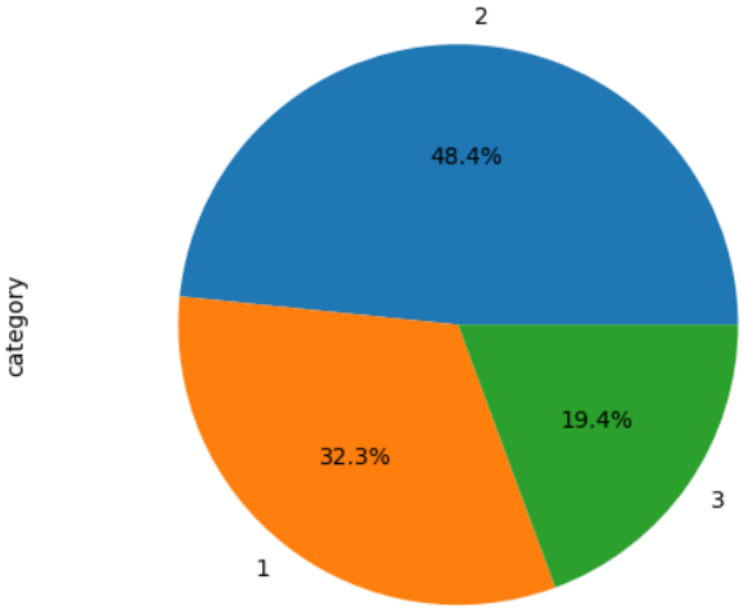
```
temp_df = df.copy()

print(f"Original dataset category proportions : {original_counts}")

for subset in needed_subsets:
    linkage_matrix = linkage(temp_df[subset], method='complete')
    add_cluster_column(temp_df,linkage_matrix)
    counts = temp_df['cluster'].value_counts()
    print(f'Cluster proproptions by {subset} : {counts}')
```

```
Original dataset category proportions : 2    150
1    100
3     60
Name: category, dtype: int64
Cluster proproptions by ['c_tilt', 'lumbar_lordosis_angle', 'pelvic_radius'] : 2    148
3     67
1     64
Name: cluster, dtype: int64
Cluster proproptions by ['sacral_slope', 'pelvic_radius', 'degree_spondylolisthesis'] : 2    156
3     62
1     61
Name: cluster, dtype: int64
```

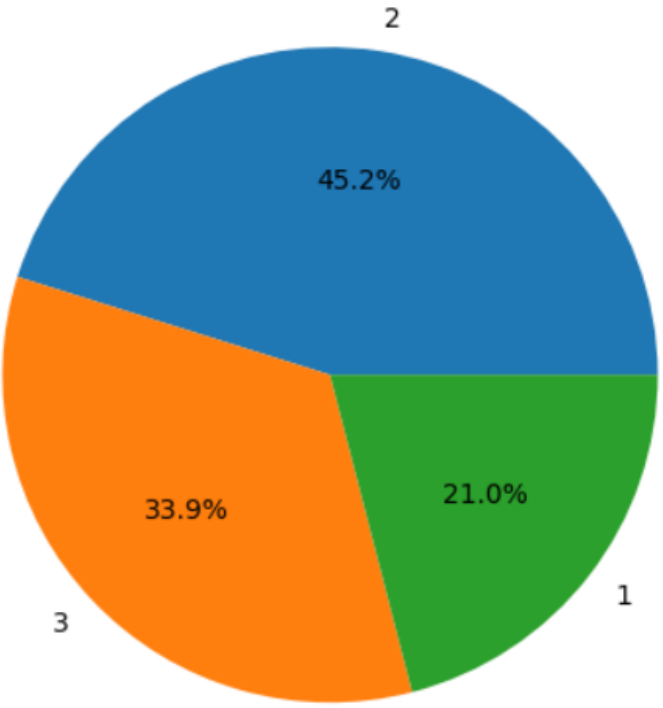
Hence, we can select [‘c_tilt’, ’lumbar_lordosis_angle’,’pelvic_radius’] or [‘sacral_slope’,’pelvic_radius’,’degree_spondylolisthesis’] as subsets for better performance.

Conclusion:

The original dataset was categorized into 3 classes: SL(2), NO(1), DH(3) with the following proportions:

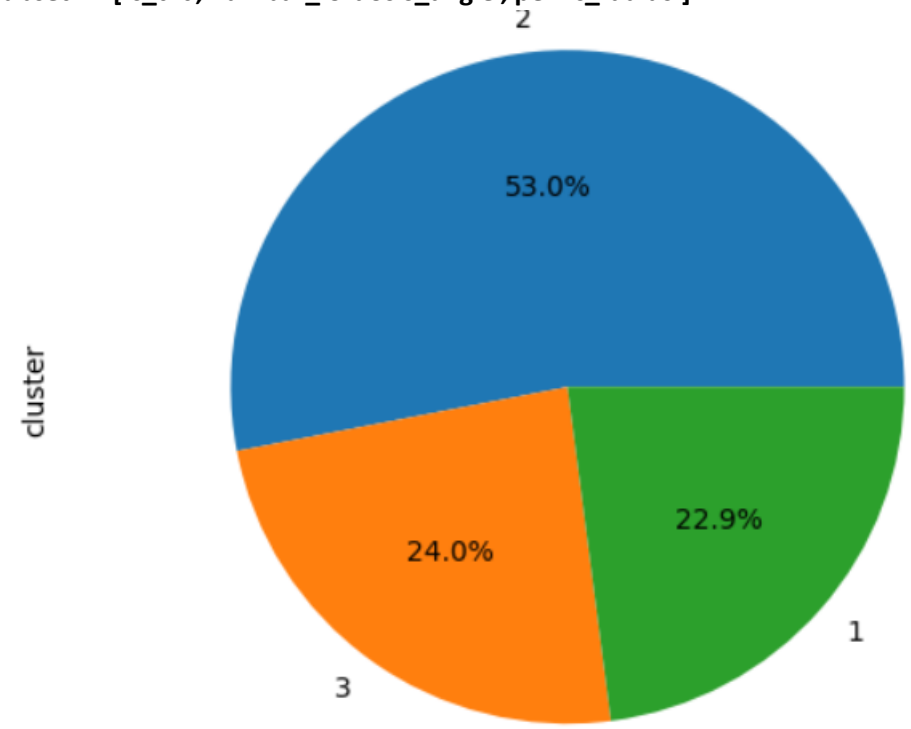


Although the outliers were not removed from the dataset – which normally causes issues – Kmeans algorithm performed well in clustering and showed almost the same proportions:



Although Hierarchical Clustering algorithm with complete linkage showed poor performance in clustering, with proper selection of subsets it almost achieved the intended grouping:

Subset1 = [‘c_tilt’, ’lumbar_lordosis_angle’,’pelvic_radius’]



Subset2 = [‘sacral_slope’,’pelvic_radius’,’degree_spondylolisthesis’]