Daniel Lupercio - STAT 724 HW5

In [ ]:
```
# %load ../standard_import.txt
import pandas as pd
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.preprocessing import scale,StandardScaler
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans
from sklearn.metrics import confusion_matrix,accuracy_score

from scipy.cluster import hierarchy
from scipy.cluster.hierarchy import dendrogram, linkage
from sklearn.cluster import AgglomerativeClustering

%matplotlib inline
plt.style.use('seaborn-white')
```

10. In this problem, you will generate simulated data, and then perform PCA and K-means clustering on the data.

Generate a simulated data set with 20 observations in each of three classes (i.e. 60 observations total), and 50 variables. Be sure to add a mean shift to the observations in each class so that there are three distinct classes.

In [ ]:
```
np.random.seed(0)
c1 = np.append(np.random.normal(0, .1, (20, 49)), np.full((20, 1), 1), axis=1)
c2 = np.append(np.random.normal(0.1, .1, (20, 49)), np.full((20, 1), 2), axis=1)
c3 = np.append(np.random.normal(-0.1, .1, (20, 49)), np.full((20, 1), 3), axis=1)

df = pd.DataFrame(np.vstack((c1,c2,c3)))
```

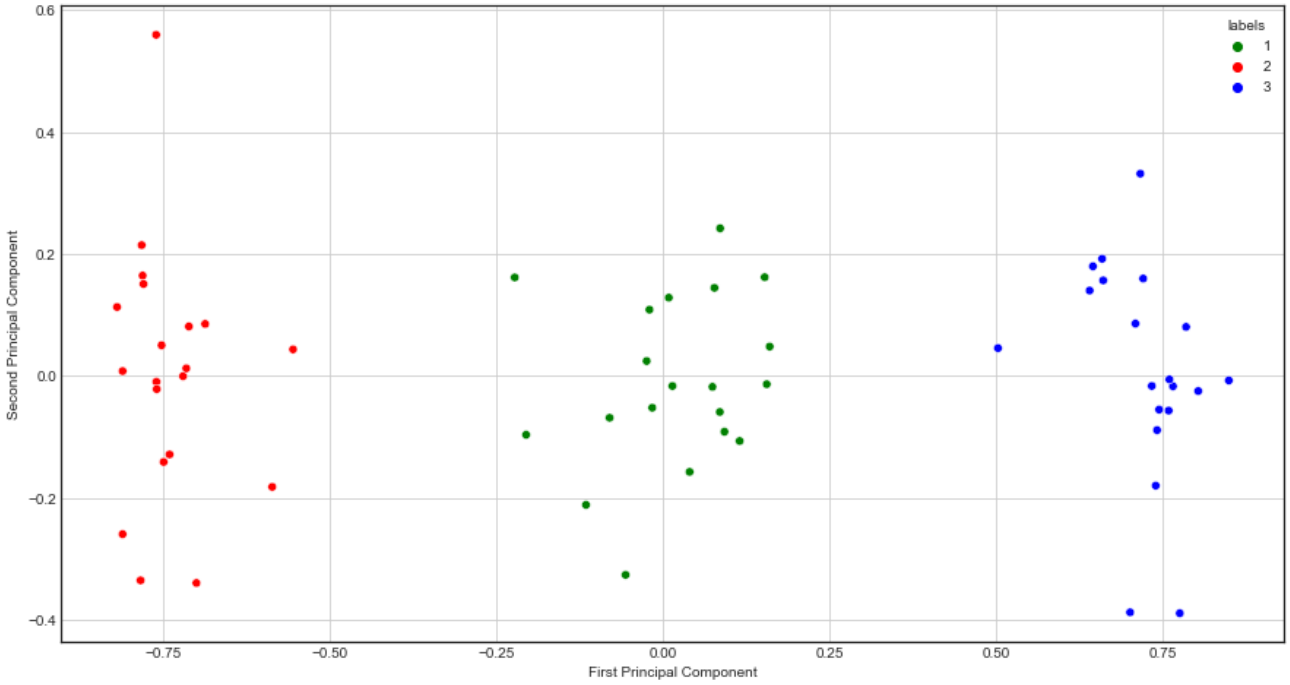
In [ ]:
```
df.shape
```

Out[ ]:
```
(60, 50)
```

(b) Perform PCA on the 60 observations and plot the first two principal component score vectors. Use a different color to indicate the observations in each of the three classes. If the three classes appear separated in this plot, then continue on to part (c). If not, then return to part (a) and modify the simulation so that there is greater separation between the three classes. Do not continue to part (c) until the three classes show at least some separation in the first two principal component score vectors.

In [ ]:
```
pca = PCA(n_components=2)
principalComponents = pca.fit_transform(df.iloc[:,0:49])
df_pca = pd.DataFrame(principalComponents, columns=['PC1', 'PC2'])
df_pca['labels'] = df[[49]]
df_pca.labels = df_pca.labels.astype(int)
```

In [ ]:
```
fig = plt.figure(figsize=(15, 8))
sns.scatterplot(x="PC1", y="PC2", hue="labels", palette={1:'green', 2:'red', 3:'blue'}, data=
plt.grid()
plt.xlabel("First Principal Component")
plt.ylabel("Second Principal Component")
plt.show()
```



(c) Perform K-means clustering of the observations with K = 3. How well do the clusters that you obtained in K-means clustering compare to the true class labels? Be careful how you interpret the results: K-means clustering will arbitrarily number the clusters, so you cannot simply check whether the true class labels and clustering labels are the same.

In [ ]:
```
km3 = KMeans(n_clusters=3, n_init=5, random_state=0)
km3.fit(df)
```

Out[ ]:
```
KMeans(n_clusters=3, n_init=5, random_state=0)
```

In [ ]:
```
y_pred = km3.fit_predict(df)
print(y_pred)
```

```
[2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

In [ ]:
```
km3_labels = (km3.labels_)
print(km3_labels)
```

```
[2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

In [ ]:
```
#Print a matrix, with class and true lables on the top and bottom
```

In [ ]:
```
pd.Series(km3.labels_).value_counts()
```

Out[ ]:
```
0    20
1    20
2    20
dtype: int64
```

In [ ]:
```
cm = confusion_matrix(y_pred, km3_labels)
print("Confusion Matrix: \n", cm)
# cm_argmax = cm.argmax(axis=0)
# print(cm_argmax)
# y_pred_ = np.array([cm_argmax[i] for i in y_pred])
print("Accuracy Score: ", accuracy_score(km3_labels,y_pred_))
```

```
Confusion Matrix:
 [[20  0  0]
```

```
 [ 0 20  0]
 [ 0  0 20]]
Accuracy Score:  1.0
```

The confusion matrix leads me to infer that the class labels are correctly classified.

(d) Perform K-means clustering of the observations with K = 2. How well do the clusters that you obtained in K-means clustering compare to the true class labels?

In [ ]:
```
km2 = KMeans(n_clusters=2, n_init=5, random_state=0)
km2.fit(df.iloc[:,:49])
```

Out[ ]:
```
KMeans(n_clusters=2, n_init=5, random_state=0)
```

In [ ]:
```
km2_labels = (km2.labels_)
print(km2_labels)
```

```
[1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
```

In [ ]:
```
y2_pred = km2.fit_predict(df)
```

In [ ]:
```
cm2 = confusion_matrix(y2_pred, km2_labels)
print("Confusion matrix : \n", cm2)
cm2_argmax = cm2.argmax(axis=0)
print(cm2_argmax)
```

```
Confusion matrix :
 [[ 0 20]
 [22 18]]
[1 0]
```

In [ ]:
```
# y2_pred_ = np.array([cm2_argmax[i] for i in y2_pred])
# cm2_ = confusion_matrix(y2_pred, km3_labels)
# print(cm2_)
print(accuracy_score(km2_labels,y2_pred))
```

```
0.3
```

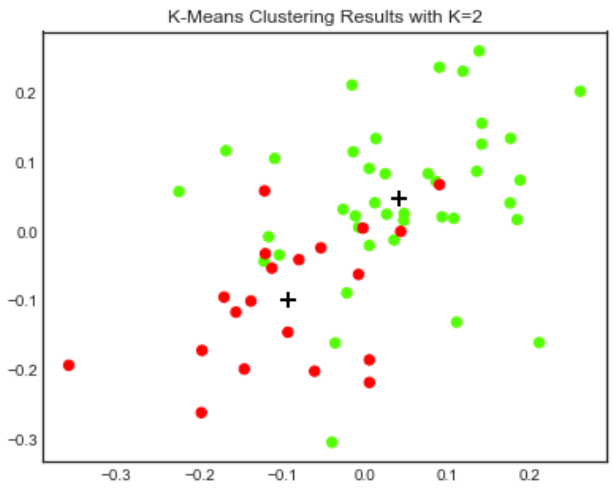
In [ ]:
```
pd.Series(km2.labels_).value_counts()
```

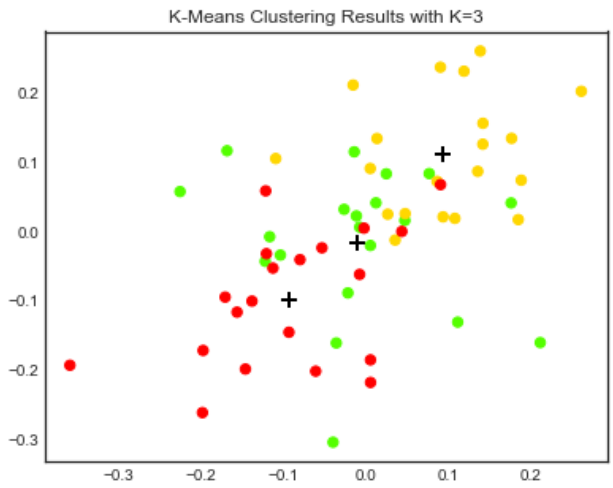
Out[ ]:
```
1    40
0    20
dtype: int64
```

In [ ]:
```
fig, (ax1, ax2) = plt.subplots(1,2, figsize=(14,5))

ax1.scatter(df.iloc[:,0], df.iloc[:,1], s=40, c=km2.labels_, cmap=plt.cm.prism)
ax1.set_title('K-Means Clustering Results with K=2')
ax1.scatter(km2.cluster_centers_[:,0], km2.cluster_centers_[:,1], marker='+', s=100, c='k', l

ax2.scatter(df.iloc[:,0], df.iloc[:,1], s=40, c=km3.labels_, cmap=plt.cm.prism)
ax2.set_title('K-Means Clustering Results with K=3')
ax2.scatter(km3.cluster_centers_[:,0], km3.cluster_centers_[:,1], marker='+', s=100, c='k', l
```





(e) Now perform K-means clustering with K = 4, and describe your results.

In [ ]:
```
km4 = KMeans(n_clusters=4, n_init=50, random_state=0)
km4.fit(df.iloc[:,:49])
```

Out[ ]:
```
KMeans(n_clusters=4, n_init=50, random_state=0)
```

In [ ]:
```
print(km4.labels_)
```

```
[0 3 3 0 0 3 3 3 0 3 0 0 3 3 0 3 3 0 0 3 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2]
```

In [ ]:
```
# Observations per KMeans cluster
pd.Series(km4.labels_).value_counts()
```

Out[ ]:
```
1    20
2    20
3    11
0     9
dtype: int64
```

The value counts functions tells us that the first two class labels were correctly clustered. However, the third class label was clustered into a subgroup.

(f) Now perform K-means clustering with K = 3 on the first two principal component score vectors, rather than on the raw data. That is, perform K-means clustering on the 60 × 2 matrix of which the first column is the first principal component score vector, and the second column is the second principal component score vector. Comment on the results.

In [ ]:
```
pca_f = PCA(n_components=3)
principalComponents = pca_f.fit_transform(df.iloc[:,0:49])
df_pca_f = pd.DataFrame(principalComponents, columns=['PC1', 'PC2', 'PC3'])
```

In [ ]:
```
df_pca_f = df_pca_f.iloc[:,0:2]
print(df_pca_f.head())
```

```
        PC1       PC2
0 -0.114929 -0.211609
1  0.009307  0.128210
2 -0.221930  0.161381
3  0.153161  0.161741
4 -0.023953  0.024275
```

In [ ]:
```
kmf = KMeans(n_clusters=3, n_init=50, random_state=0)
kmf.fit(df_pca_f)
kmf_labels = (kmf.labels_)
print(kmf_labels)
```

```
[2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

In [ ]:
```
y_f = kmf.fit_predict(df_pca_f)
```

In [ ]:
```
print("Confusion matrix: \n", confusion_matrix(y_f,kmf_labels))
```

```
Confusion matrix:
 [[20  0  0]
 [ 0 20  0]
 [ 0  0 20]]
```

In [ ]:
```
pd.Series(kmf.labels_).value_counts()
```

Out[ ]:
```
0    20
1    20
2    20
dtype: int64
```

It appears the labels are matched perfectly, based on the value counts. The two principal components were able to reduce dimensional space without loss of information.

(g) Using the scale() function, perform K-means clustering with K = 3 on the data after scaling each variable to have standard deviation one. How do these results compare to those obtained in (b)? Explain.

In [ ]:
```
#omit the mean parameter
data_scaler = StandardScaler(with_mean= False, with_std=True)
df_scaled = data_scaler.fit_transform(df)
```

In [ ]:
```
km_g = KMeans(n_clusters=3,n_init=5, random_state=0)
km_g.fit(df_scaled)
km_g_labels = (km_g.labels_)
print(km_g_labels)
```

```
[2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

In [ ]:
```
y_g = km_g.fit_predict(df_scaled)
```

In [ ]:
```
print("Confusion Matrix: \n", confusion_matrix(km_g_labels, y_g))
```

```
Confusion Matrix:
 [[20  0  0]
 [ 0 20  0]
 [ 0  0 20]]
```

In [ ]:
```
print("Accuracy score: ", accuracy_score(km_g_labels,y_g))
```

```
Accuracy score:  1.0
```

In [ ]:
```
pd.Series(km_g.labels_).value_counts()
```

Out[ ]:
```
0    20
1    20
2    20
dtype: int64
```

As with the results from (f), the labels are matched perfectly. But this can not be interpreted directly, as scaling the dataframe can alter the results of the k-means algorithm.

13. On the book website, www.statlearning.com, there is a gene expression data set (Ch12Ex13.csv) that consists of 40 tissue

samples with measurements on 1,000 genes. The first 20 samples are from healthy patients, while the second 20 are from a diseased group.

(a) Load in the data using read.csv(). You will need to select header = F.

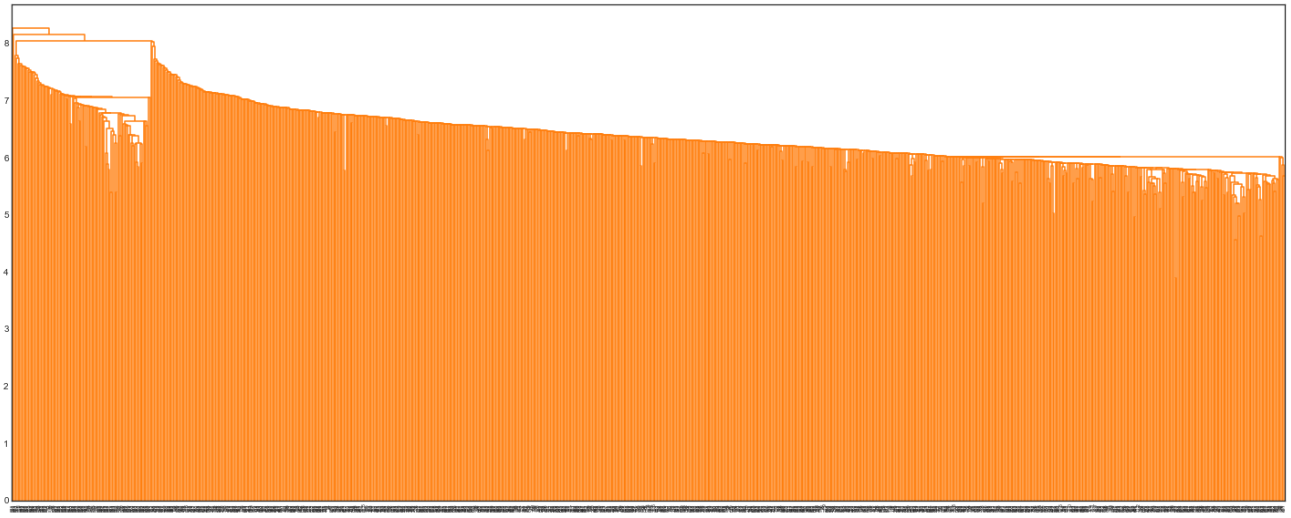
In [ ]:
```
gene_df = pd.read_csv('/Users/daniel421/Desktop/STAT_724/ISLR_data/Ch12Ex13.csv', header=None
# print(gene_df.head())
```

(b) Apply hierarchical clustering to the samples using correlation based distance, and plot the dendrogram. Do the genes separate the samples into the two groups? Do your results depend on the type of linkage used?

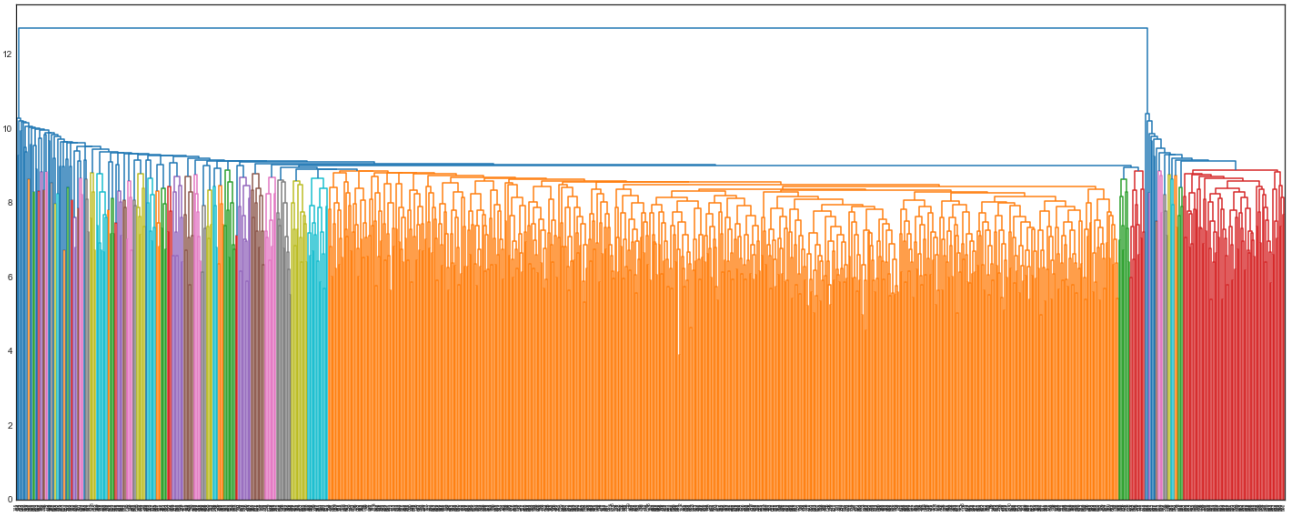
In [ ]:
```
# hierarchy.dendrogram(hierarchy.complete(gene_df), truncate_mode='lastp', p=4, show_leaf_cou
```

https://docs.scipy.org/doc/scipy/reference/generated/scipy.cluster.hierarchy.linkage.html

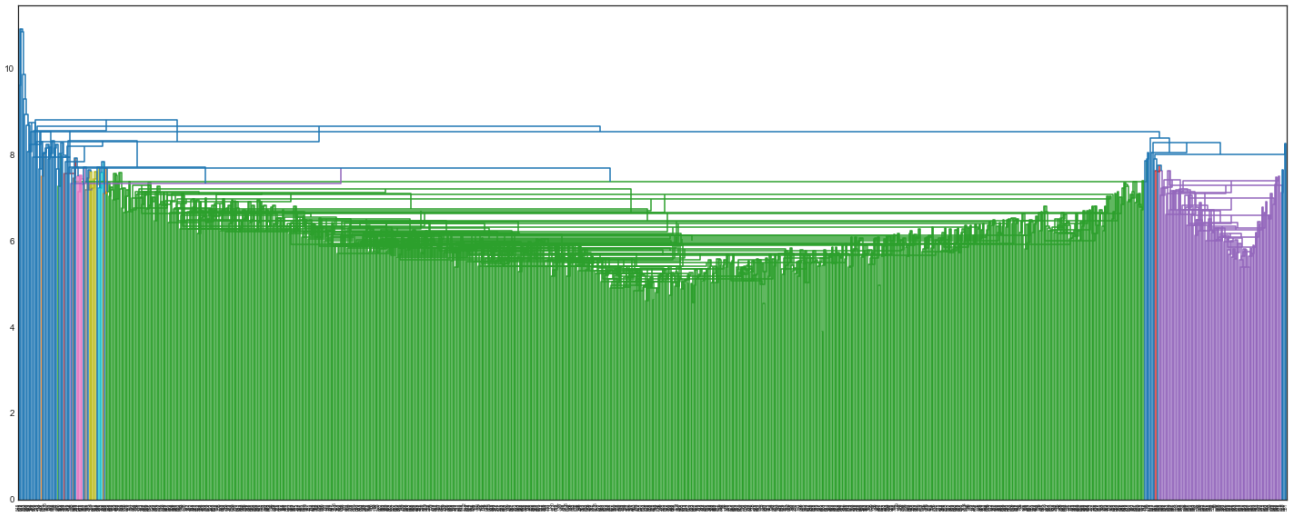
In [ ]:
```
Z = linkage(gene_df, 'single')
fig = plt.figure(figsize=(25,10))
dn = dendrogram(Z)
plt.show()
```



In [ ]:
```
Z2 = linkage(gene_df, 'average')
fig = plt.figure(figsize=(25,10))
dn = dendrogram(Z2)
plt.show()
```



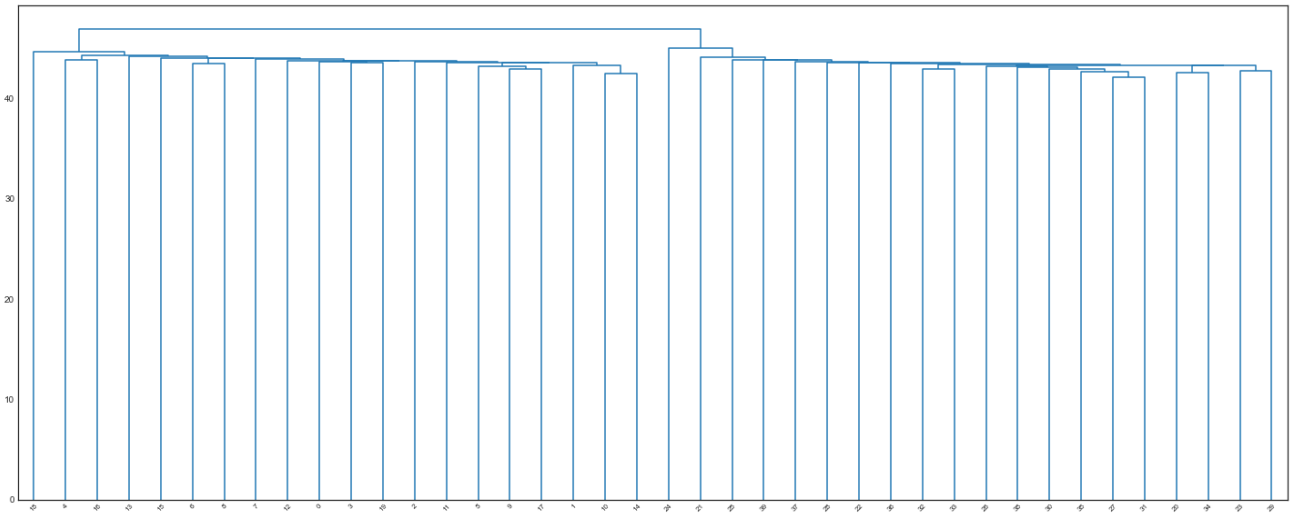
In [ ]:
```
Z3 = linkage(gene_df, 'median')
fig = plt.figure(figsize=(25,10))
dn = dendrogram(Z3)
plt.show()
```



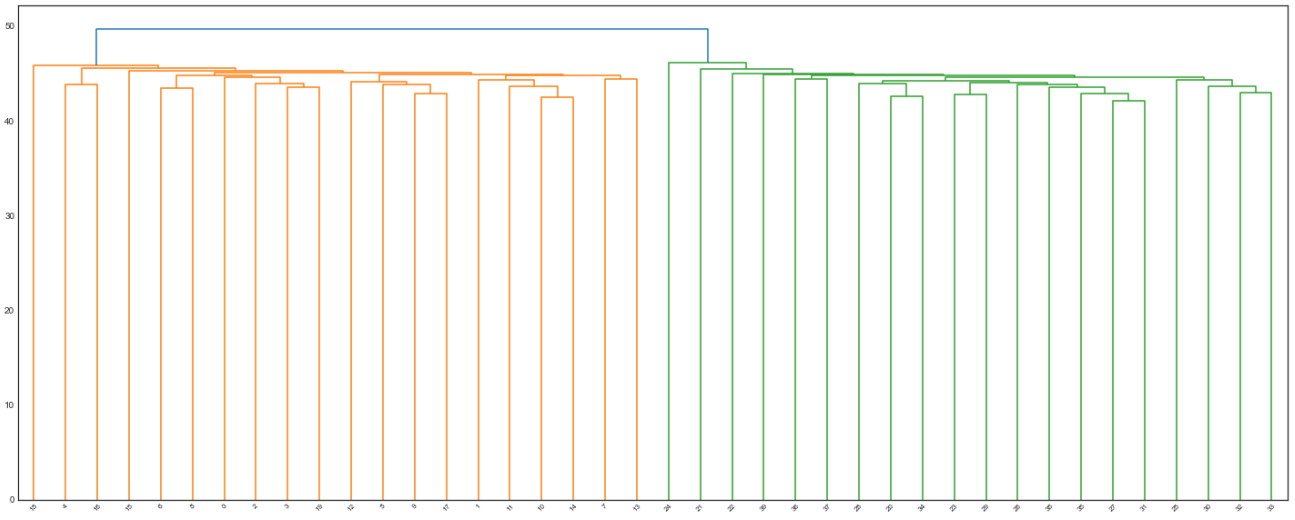
These results are un-iterpretable, I will transpose the gene dataframe.

In [ ]:
```
gene_df2 = gene_df.T
# print(gene_df2.head())
```

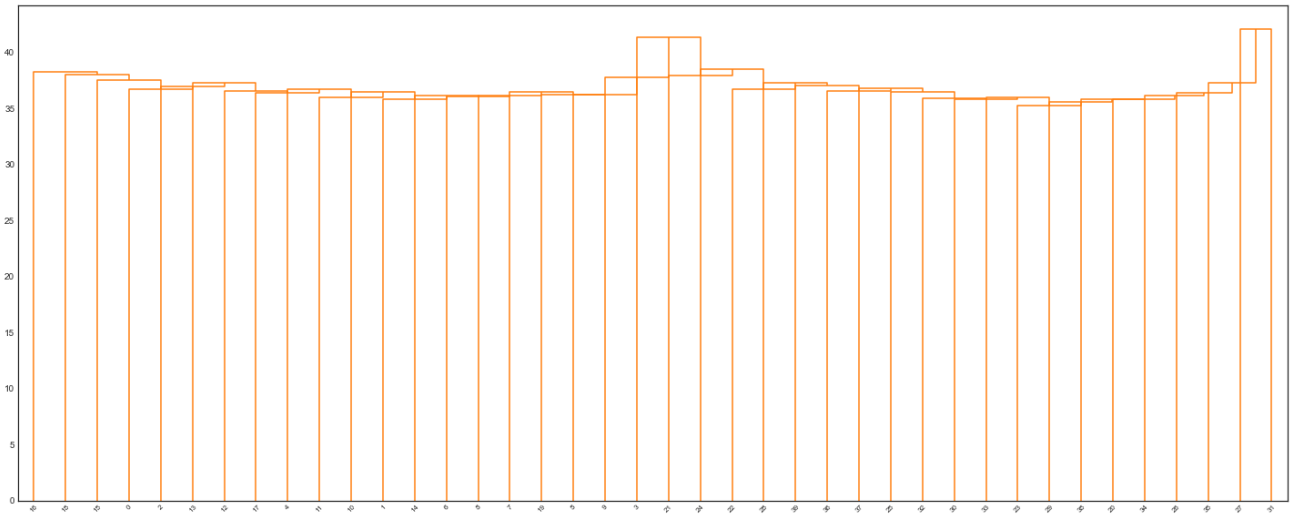
In [ ]:
```
Z = linkage(gene_df2, 'single')
fig = plt.figure(figsize=(25,10))
dn = dendrogram(Z)
plt.show()
```



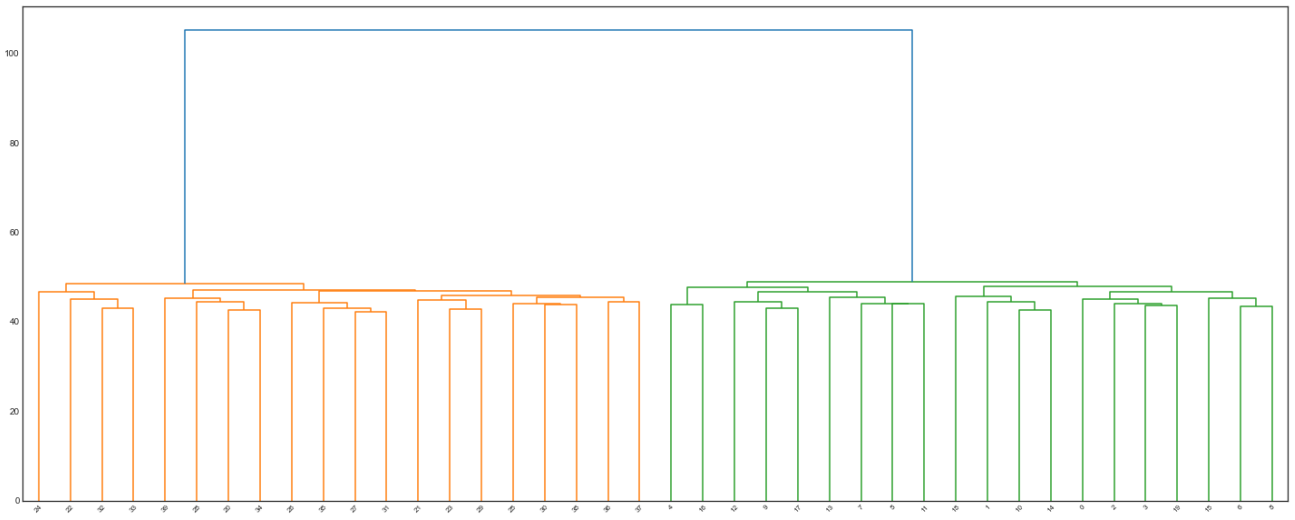
In [ ]:
```
Z2 = linkage(gene_df2, 'average')
fig = plt.figure(figsize=(25,10))
dn = dendrogram(Z2, color_threshold = 49)
plt.show()
```



In [ ]:
```
Z3 = linkage(gene_df2, 'median')
fig = plt.figure(figsize=(25,10))
dn = dendrogram(Z3, color_threshold=49)
plt.show()
```

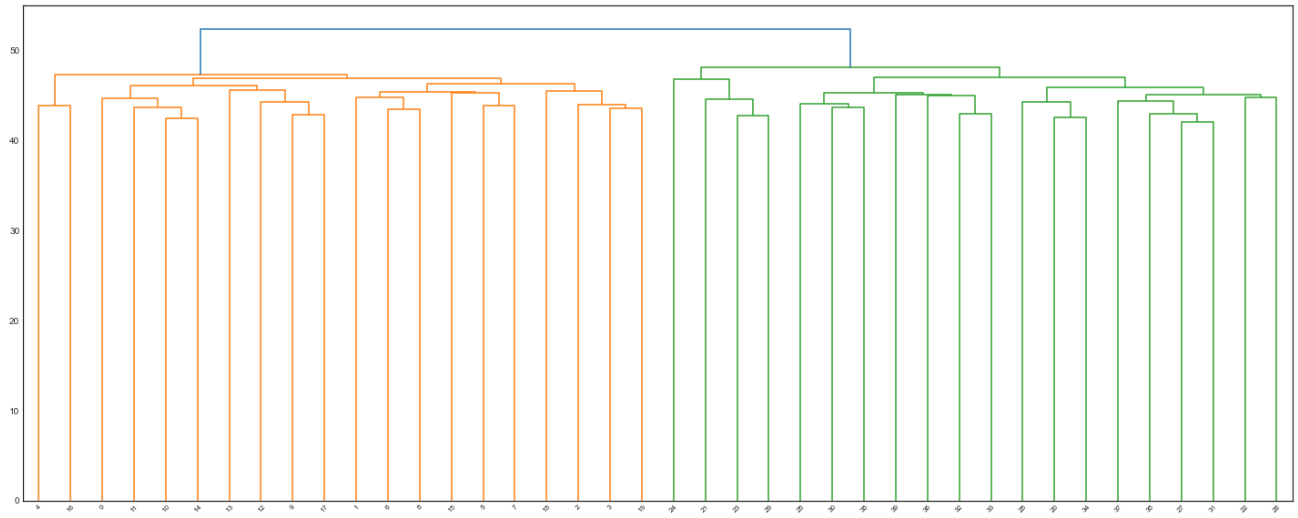


In [ ]:
```
Z4= linkage(gene_df2, 'ward')
fig = plt.figure(figsize=(25,10))
dn = dendrogram(Z4)
plt.show()
```



In [ ]:
```
Z5 = linkage(gene_df2, 'complete')
fig = plt.figure(figsize=(25,10))
```

```
dn = dendrogram(Z5,color_threshold=49)
plt.show()
```



It appears that most linkage methods do seperate the genes into two groups.

(c) Your collaborator wants to know which genes differ the most across the two groups. Suggest a way to answer this question, and apply it here.

Perhaps we can implement principal component analysis.

In [ ]:
```
pca_gene = PCA(n_components=40)
principalComponents = pca_gene.fit(gene_df)
# df_pca_gene = pd.DataFrame(principalComponents, columns=['PC1', 'PC2'])
# df_pca['labels'] = gene_df2[[1000]]
# df_pca.labels = df_pca.labels.astype(int)
```

In [ ]:
```
print(principalComponents.explained_variance_ratio_)
```

```
[0.18699299 0.02998891 0.02842005 0.02751355 0.02714304 0.02649283
 0.02585679 0.0253583  0.02489026 0.02467986 0.02431694 0.02423387
 0.02355519 0.02297315 0.02263365 0.02243292 0.02177749 0.02151727
 0.02111598 0.02088552 0.02060847 0.02035784 0.01995302 0.01982156
 0.01934277 0.0188597  0.01867544 0.01805316 0.0178497  0.01766985
 0.0173848  0.01691103 0.01659561 0.01647373 0.01588294 0.01531126
 0.01525017 0.01461167 0.01382375 0.01378497]
```

We can see that the majority of the variance is explained by one principal component. The rest of the variance is small and is scattered with the othe 39 principal components.