B565-Data Mining
Homework #5

Due on Friday, March 12, 2023, 08:00 p.m.

Instructor: Dr. H. Kurban, Head TA: Md R. Kabir

Prem Amal

March 12, 2023

PCA Algorithm

This part is provided to help you implement Principal Component Analysis.

1: **ALGORITHM** `PCA`
2: **INPUT** Δ: $n \times m$ data matrix of rank r, d: the number of new dimensions where $d \leq r$.
3: **OUTPUT** $\hat{\Delta}$: d dimensional representation of Δ.
4: %% mean centering - $\tilde{\Delta}$ is the centered data matrix.
5: %% I denotes $n \times n$ identity matrix, $e = (1, ..., 1) \in \Re^n$.
6: $\tilde{\Delta} \leftarrow \left(I - \frac{ee^t}{n}\right)\Delta$.
7: %% Compute the SVD of $\tilde{\Delta}$. $\sigma_1 \geq ... \geq \sigma_r > 0$ are the strictly positive singular values of $\tilde{\Delta}$.
8: %% $u_1, ..., u_r$ and $v_1, ..., v_r$ are the corresponding left and right singular vectors respectively.
9: $\tilde{\Delta} \leftarrow \sum_{i=1}^{r} \sigma_i u_i v_i^t$
10: $\hat{\Delta}\left[\sigma_1 u_1 | ... | \sigma_d u_d\right] = \begin{bmatrix} \hat{\delta}_1^t \\ \vdots \\ \hat{\delta}_n^t \end{bmatrix}$

Problem 1

Implement Principal Component Analysis algorithm (PCA) and run your program over RNA-Seq (HiSeq) PANCAN data set (the data contains the gene expressions of patients having 5 different types of tumor: BRCA, KIRC, COAD, LUAD and PRAD which we will be using as our data labels) to answer the following questions. **Data Preparation:** Similar to any other data mining problem, before feeding your data to the clustering algorithms, you will have to perform appropriate data cleaning, feature engineering, and feature selection on this dataset **[25 pt.]**

The dataset contains the gene expression values for 20,531 genes across 800 cancer samples. Additionally, a CSV file called 'labels.csv' indicates the type of cancer for each of the 800 data points. This data has been obtained from the TCGA project.
The purpose of dataset can be to perform various analyses related to cancer biology, such as identifying biomarkers for different types of cancer,etc.
This dataset can be used by researchers to develop predictive models to identify patients who may be at a higher risk of developing certain types of cancer.

1. Perform PCA over the RNA-Seq data set and make a scatter plot of PC1 and PC2 (the first two principal components). Are PC1 and PC2 linearly correlated? How much of the variance can be explained by PC1 and PC2?

R or Python script

```r
# Sample R Script With Highlighting
```

```python
# Sample Python Script With Highlighting
import pandas as pd
import swifter
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import LabelEncoder
from tqdm import tqdm
import numpy as np
```

```
from sklearn.preprocessing import StandardScaler
from numpy import linalg as LA
import warnings


df_data= pd.read_csv('TCGA-PANCAN-HiSeq-
801x20531/data.csv')
df_label=pd.read_csv('TCGA-PANCAN-HiSeq-
801x20531/labels.csv')
df_data.head()

Unnamed: 0     gene_0    gene_1    gene_2    gene_3    gene_4    gene_5    gene_6    gene_7    g
0    sample_0  0.0  2.017209  3.265527  5.478487  10.431999 0.0  7.175175  0.591871  0.0  ...  4
1    sample_1  0.0  0.592732  1.588421  7.586157  9.623011  0.0  6.816049  0.000000  0.0  ...  4
2    sample_2  0.0  3.511759  4.327199  6.881787  9.870730  0.0  6.972130  0.452595  0.0  ...  5
3    sample_3  0.0  3.663618  4.507649  6.659068  10.196184 0.0  7.843375  0.434882  0.0  ...  6
4    sample_4  0.0  2.655741  2.821547  6.539454  9.738265  0.0  6.566967  0.360982  0.0  ...  5
5 rows    20532 columns

#Here the column 'Unnamed: 0' seems to be the primary key, #indicating which sample is the row.
df_data.drop(columns=['Unnamed: 0'],inplace=True)

df_label.head()

#Here the column 'Unnamed: 0' seems to be the primary key, #indicating which sample is the row.

df_label.drop(columns=['Unnamed: 0'],inplace=True)

#Checking for any null values in the dataframe
null_feature=[i for i in df_data.columns if df_data[i].isnull().sum()>=1]
print(null_feature)
[]
null_feature=[i for i in df_label.columns if df_label[i].isnull().sum()>=1]
print(null_feature)
[]
#No columns have null data

#PCA

def PCA(df,threshold):
    ## Performing Standardization so that all features are
    given same importance initially and each feature can
    contribute
    ## equally to PC irrespective of scale or magnitude
    df =pd.DataFrame(StandardScaler().fit_transform(df))
    ## Performing Data Centering on standardized dataframe
    centred_df= df-np.mean(df,axis=0)
    ## Calculating Covariance
    covariance=np.cov(df.T)
    eigen_values, eigen_vectors = LA.eig(covariance)
    ## Sorting the eigen values in descending order, using
    argsort, we get the indices of eigen values in
    descending order
```

```
    sorted_index=eigen_values.argsort()[::-1]
    df_variance=pd.DataFrame(eigen_values[sorted_index]/sum
    (eigen_values),columns=['variance'])
    df_variance['cumulative_variance']=
    df_variance['variance'].cumsum()
    ## Number of principal components required to cover
    variance uptill certian threshold

    df_number_pc_var=df_variance[df_variance['cumulative_va
    riance']<= threshold]
    number_of_pc=len(df_number_pc_var)
    print("The number of principal components required to
    cover {} percent variance are
    {}".format(threshold*100,number_of_pc))
    ## Selecting the required number of eigen vectores for performing dot product

    selected_eigen_vectors=eigen_vectors[:,sorted_index[:nu
    mber_of_pc]]

    #Projecting data over the selected number of principal components
    principal_component=centred_df.dot(selected_eigen_vecto
    rs) #Performing dot product
    principal_component.columns=[f'PC{i+1}' for i in
    range(number_of_pc)]
    ## Creating Scree plots

    fig, axes = plt.subplots(1, 2, figsize=(20, 6))

    # Plot the first subplot of variance on the left side
    axes[0].plot(range(1,len(df_number_pc_var)+1),
    df_number_pc_var['variance'],'ro-', linewidth=2)
    axes[0].set_title('Scree Plot for Variance')
    axes[0].set_ylabel('Variance')
    axes[0].set_xlabel('Number of Principal Component')


    # Plot the second subplot of cumulative variance on
    the right side
    axes[1].plot(range(1,len(df_number_pc_var)+1),
    df_number_pc_var['cumulative_variance'],'ro-',
    linewidth=2)
    axes[1].set_title('Scree Plot for Cumulative Variance')
    axes[1].set_ylabel('Cumulative Variance')
    axes[1].set_xlabel('Number of Principal Component')

    plt.show()
    print('Correlation of PC1,PC2
    \n',principal_component[['PC1','PC2']].corr())
    plt.scatter(principal_component['PC1'],principal_compon
    ent['PC2'])
    plt.xlabel('PC1')
    plt.ylabel('PC2')
    plt.title('Scatter plot of PC1 VS PC2')
```

```
    plt.show()
    print('The Total variance explained by PC1,PC2 is {} percent'.format(round(np.real(df_number

    loadings =
    pd.DataFrame(selected_eigen_vectors,index=df.columns)
    warnings.filterwarnings("ignore")

    return
    principal_component,loadings,df_variance,df_number_pc_v
    ar,eigen_values, eigen_vectors

principal_component,loadings,df_variance,df_number_pc_var,e
igen_values, eigen_vectors=PCA(df_data,0.9)

#The number of principal components required to cover 90.0
#percent variance are 372

Correlation of PC1,PC2
               PC1           PC2
PC1  1.000000e+00 -3.620830e-15
PC2 -3.620830e-15  1.000000e+00

#The Total variance explained by PC1,PC2 is 19.29 percent
```

Discussion of Experiments

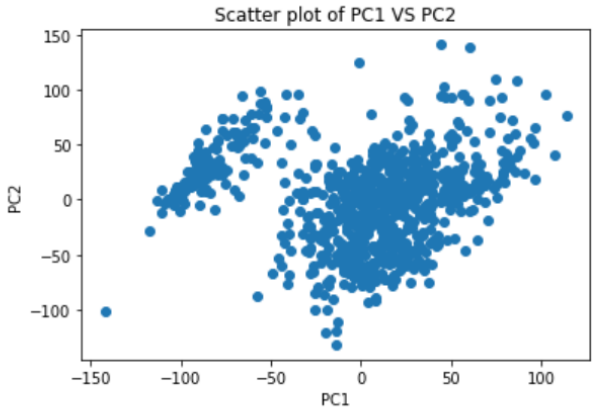
Answer here...

Have performed PCA on the RNA dataset. Please find the the scatter plot below. The points in the scatter plot are randomly spread out. There is no pattern in the scatter plot for PC1 and PC2, depicting no such relationship.
This is confirmed by the correlation matrix plotted below. The value is $-3.62 * 10^-15$, very small, showing a very weak correlation among the first two Principal Components.
Hence, PC1 and PC2 are not correlated.
The total variance explained by PC1 and PC2 is 19.29 percent.

Plot/s

Place images here with suitable captions.

```
Correlation of PC1,PC2
                PC1           PC2
PC1  1.000000e+00 -3.620830e-15
PC2 -3.620830e-15  1.000000e+00
```




```
The Total variance explained by PC1,PC2 is 19.29 percent
```

(1)

2. There are three methods to pick the set of principle components: (1) In the plot where the curve bends; (2) Add the percentage variance until total 75% is reached (70 – 90%) (3) Use the components whose variance is at least one. Show the components selected in the RNA-Seq data set data if each of these is used.

R or Python script

```r
# Sample R Script With Highlighting
```

```python
# Sample Python Script With Highlighting
fig, axes = plt.subplots(1, 2, figsize=(20, 6))

    # Plot the first subplot of variance on the left side
    axes[0].plot(range(1,len(df_number_pc_var)+1),
    df_number_pc_var['variance'],'ro-', linewidth=2)
    axes[0].set_title('Scree Plot for Variance')
    axes[0].set_ylabel('Variance')
    axes[0].set_xlabel('Number of Principal Component')


    # Plot the second subplot of cumulative variance on
    the right side
    axes[1].plot(range(1,len(df_number_pc_var)+1),
    df_number_pc_var['cumulative_variance'],'ro-',
    linewidth=2)
    axes[1].set_title('Scree Plot for Cumulative Variance')
    axes[1].set_ylabel('Cumulative Variance')
    axes[1].set_xlabel('Number of Principal Component')
```

```
    plt.show()

    df_variance_r=
    pd.DataFrame(np.real(df_variance.values),
    columns=df_variance.columns)
variance_limit=[float(i)/100 for i in range(70,91)]
nom_pc=[]
for i in variance_limit:
    nom_pc.append(len(df_variance_r[df_variance_r['cumula
    tive_variance']<= i]))

plt.plot(nom_pc,variance_limit)
plt.xlabel('Number of principal component')
plt.ylabel('Variance')
plt.title('Percentage variance')
plt.show()
plt.scatter(loadings[0],loadings[1])
plt.xlabel('PC1 Loading')
plt.ylabel('PC2 Loading')
plt.title('Scatter plot of loading PC1 VS PC2')
plt.show()
```

Discussion of Experiments

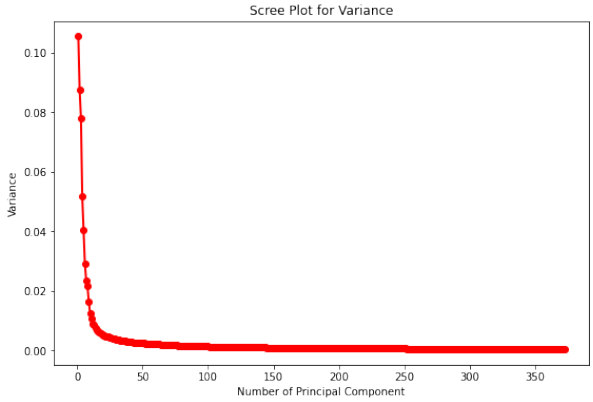
Answer here...

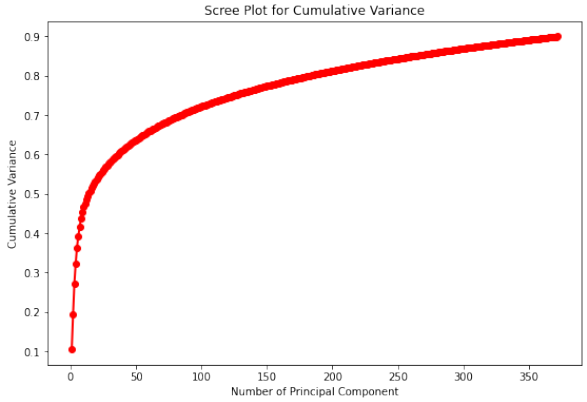
From the below plots of variance and cumulative variance it can be seen that the curve starts bending at number of principal components equals 25-30. From the plot of cumulative variance, it can be seen that after 350 number of components the plot is almost a straight , showing not much change in variance captured with increase in number of components. From the variance graph, it can be seen that after 100 number of components there is no much change in variance, hence the number of components can be anywhere from 25 till 100, the point where curve bends. To get more and more variance captured, we can select a higher number of components aswell. This depends upon the use case.
For the second method, for taking 75percent of variance, have plotted the graph showing the number of components vs the variance. Also, a list is shown in output giving the variance percent and the number of components. Have kept variance threshold from 0.7 till 0.9 with increase of 0.01 at each iteration and counted the number of principal components required.For 75percent variance, 124 principal components are required.Graph conveys the same results, number of principal components increases with increase in variance.
For 3rd method, have filtered the eigen values giving value greater than one, we get 790-800 number of principal components.

The choice of the best method for selecting the number of principal components depends on the specific dataset and analysis goals. There are three common methods for selecting the number of principal components: visual inspection of the scree plot, selecting a threshold percentage of total variance explained, and retaining components with variance at least one. Each method has strengths and weaknesses, and the best method for a given problem may vary depending on the data and research questions.

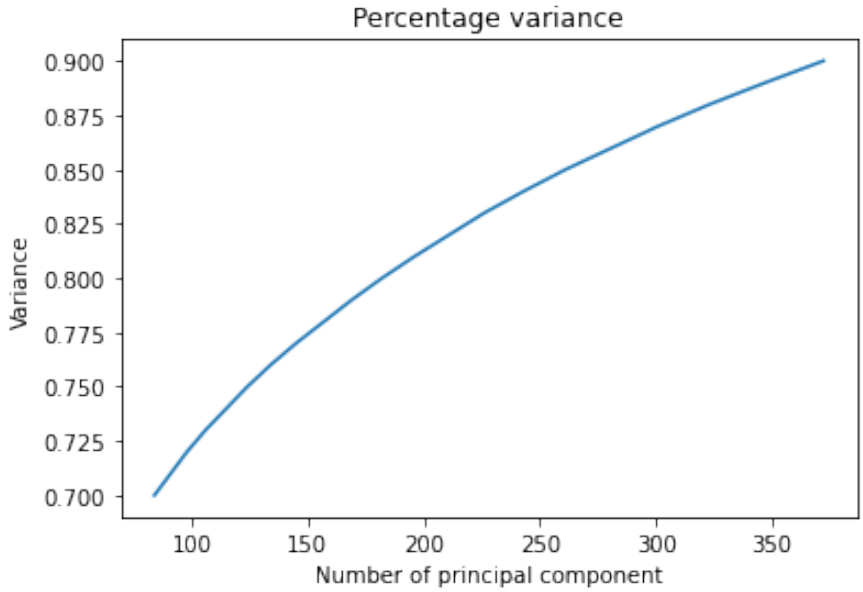
Plot/s

Place images here with suitable captions.





(2)



(3)

3. Observe & discuss the loadings in PCA? (e.g., how are principal components and original variables related?)

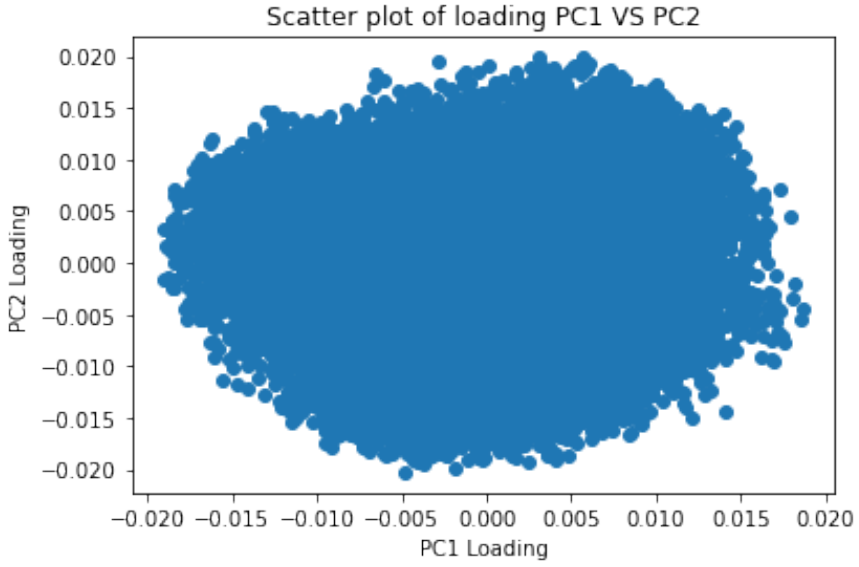
Discussion of Experiments

Answer here...
Loadings in PCA indicate the correlation between original variables and principal components. They show the relationship strength and direction. Each principal component is a linear combination of original variables, with loadings indicating their relative importance.
The first principal component accounts for the most variation and each successive component accounts for remaining variation. Loadings of the first component help to identify variables contributing most

to data variation. They also help detect multicollinearity issues, where highly correlated variables may have similar loadings.
Removing variables may be necessary to prevent over- representation. Loadings provide insight into the relationship between original variables and principal components and help interpret data.Here the scatter plot of PC1 and PC2 is shown in the figure below. Contribution of every variable to the formation of PCA is ranging from -0.02 to 0.02. The values seems randomly spread out forming a cluster and even the values are 0, showing no contribution. There are variables which are not contributing to the Principal components.

Plot/s

Place images here with suitable captions.



(4)

4. Perform dimensionality reduction over the RNA-Seq data set with PCA. Keep 90% of variance after PCA and reduce the RNA-Seq data set and call this data Δ_R. Cluster Δ_R using your k-means program from previous assignment and report the total error rates for $k = 5$ and 20 runs. Plots are generally a good way to convey complex ideas quickly, i.e., box plots, whisker plots. Discuss your results, i.e.,Did clustering get better after PCA?

R or Python script

```
# Sample R Script With Highlighting
```

```
# Sample Python Script With Highlighting

import pandas as pd
import swifter
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import LabelEncoder
from tqdm import tqdm
import numpy as np
```

```
from sklearn.preprocessing import StandardScaler
from numpy import linalg as LA
import warnings
from scipy.spatial.distance import euclidean


df_data= pd.read_csv('TCGA-PANCAN-HiSeq-
801x20531/data.csv')
df_label=pd.read_csv('TCGA-PANCAN-HiSeq-
801x20531/labels.csv')
df_data.drop(columns=['Unnamed: 0'],inplace=True)
df_label.drop(columns=['Unnamed: 0'],inplace=True)
df_label['Class'] =
LabelEncoder().fit_transform(df_label['Class'])

df= df_data.join(df_label)
df

import time
def initialize_centroids(df, k):
    """
    Function to initialize random centroids from dataset.
    Input:
        - df: pandas dataframe with the data
        - k: integer number of clusters
    Output:
        - temp_df: pandas dataframe with the centroids as
        columns and index as label
    """
    centroids = []
    for i in range(k):
        centroids.append(df.apply(lambda x:
        float(x.sample()))) # Take a random sample from
        each column to create a centroid
    centroids = pd.concat(centroids, axis=1)
    centroids.index.name = 'Label'

    return centroids


def assign_labels(df, centroids):
    """
    Function to calculate the closest centroid label for
    each row in a dataframe.
    Input:
        - df: pandas dataframe with the data
        - centroids: pandas dataframe with the centroids
        as columns and index as label
    Output:
        - distances.idxmin(axis=1): pandas series with the
        label of the closest centroid for each row in df
    """
    distances = centroids.swifter.apply(lambda x:
```

```
    np.sqrt(((df - x) ** 2).sum(axis=1))) # Calculate the
    Euclidean distance between each row in df and each
    centroid
    return distances.idxmin(axis=1) # Get the index of the
    minimum distance, which corresponds to the label of
    the closest centroid


def new_centroids(df_label, df1):
    """
    Function to calculate the new centroids based on the
    current labels of the rows.
    Input:
        - df_label: pandas series with the label of the
        closest centroid for each row in df1
        - df1: pandas dataframe with the data
    Output:
        - new_centroids.T: pandas dataframe with the new
        centroids as columns and index as feature name
    """
    joined_df = df1.join(df_label)
    joined_df.rename(columns={0: 'Label'}, inplace=True) #
    Rename the column with the label
    # Calculate the mean of the rows with the same label
    return joined_df.groupby('Label').mean().T # Transpose
    the dataframe to have the new centroids as columns and
    index as feature name


def sum_of_square_error(new_centroids, data, labels):
    """
    Computes the sum of squared errors between the data
    points and their assigned centroids.

    Args:
    new_centroids (DataFrame): The new centroids computed
    in the current iteration.
    data (DataFrame): The input data points.
    labels (DataFrame): The labels assigned to each data
    point.

    Returns:
    The sum of squared errors.
    """
    # Transpose the new centroids dataframe and reset the
    index
    new_centroids = new_centroids.T.reset_index()
    # Get the columns of the data dataframe
    columns = data.columns
    # Join the data dataframe and the labels dataframe
    data = data.join(labels)
    # Rename the '0' column of the labels dataframe to
    'Label'
```

```
    data.rename(columns={0:'Label'}, inplace=True)
    sse = []
    # Compute the distance between each data point and its
    assigned centroid
    for i in range(len(new_centroids)):
        distance = np.sum(np.square(data[data['Label']==i]
        [columns] - new_centroids.iloc[i][columns]),
        axis=1)
        sse.append(sum(distance))
    # Return the sum of squared errors
    return sum(sse)

def kmeans_lyod_with_error(df1, k, tou):
    """
    Function to run the K-means Lloyd algorithm.
    Input:
        - df1: pandas dataframe with the data
        - k: integer number of clusters
        - tou: float tolerance level to stop the algorithm
    Output:
        - centroids: pandas dataframe with the final centroids as columns and index as label
    """
    start_time=time.time()
    centroids = initialize_centroids(df1, k) # Initialize
    random centroids
    initial_list_of_columns = centroids.columns.to_list()
    iteration = 0
    while True:
        # Assign labels to current centroids
        df_label = assign_labels(df1, centroids)
        df_label = pd.DataFrame(df_label)
        # Calculate new centroids
        df_new_centroids = new_centroids(df_label, df1)
        new_list_of_columns =
        df_new_centroids.columns.to_list()
        # Keep the number of clusters the same i.e
        maintain same k
        for i in initial_list_of_columns:
            if i not in new_list_of_columns:
                df_new_centroids[i] = centroids[i]
        # Calculate tao
        distance = []
        for col in centroids.columns:
            col_distance = euclidean(centroids[col],
            df_new_centroids[col])
            distance.append(col_distance)
        tao_calculated=sum(distance)/k #Used the formula
        provided for calculating Tao
        sse = sum_of_square_error(df_new_centroids, df1,
        df_label)
        #error=error_clusters(df_label,df1,k)
        end_time= time.time()
        if iteration>100:
```

```
            print("Iteration exceeded")

            return  sse,end_time-start_time
            break

        if tao_calculated<tou or iteration >100:   #if the
        convergence is met, kmeans will stop  or else if
        the convergence is never met, after 100 iteration
        code will stop
            return  sse,end_time-start_time
            break                                  #
            otherwise indefinite loop
        else:
            centroids= df_new_centroids # In case we need
            more iterations, the centroids calculated at
            this step acts as input
        iteration+=1


error_matrix=[]
for i in range (1,21):
    sse,run_time=kmeans_lyod_with_error(df,5,10)
    error_matrix.append([i,sse,run_time])
error_matrix_df= pd.DataFrame(error_matrix,columns=[
'repetition','sse','run_time'])
error_matrix_df

### After clustering

principal_component=pd.read_csv('principal_component.csv')
principal_component=principal_component.join(df_label)
principal_component

error_matrix=[]
for i in range (1,21):

    sse,run_time=kmeans_lyod_with_error(principal_component
    ,5,10)
    error_matrix.append([i,sse,run_time])
error_matrix_df_pca= pd.DataFrame(error_matrix,columns=[
'repetition','sse','run_time'])
error_matrix_df_pca.to_csv('error_matrix_df_pca.csv')

error_matrix_df_pca['sse'].mean()
#11465730.680092001
error_matrix_df['sse'].mean()
#19045435.84364903

combined_df = pd.concat([error_matrix_df_pca,
error_matrix_df], keys=['after_pca', 'before_pca'])

# create the boxplot
plt.boxplot([combined_df.loc['after_pca']['sse'],
```

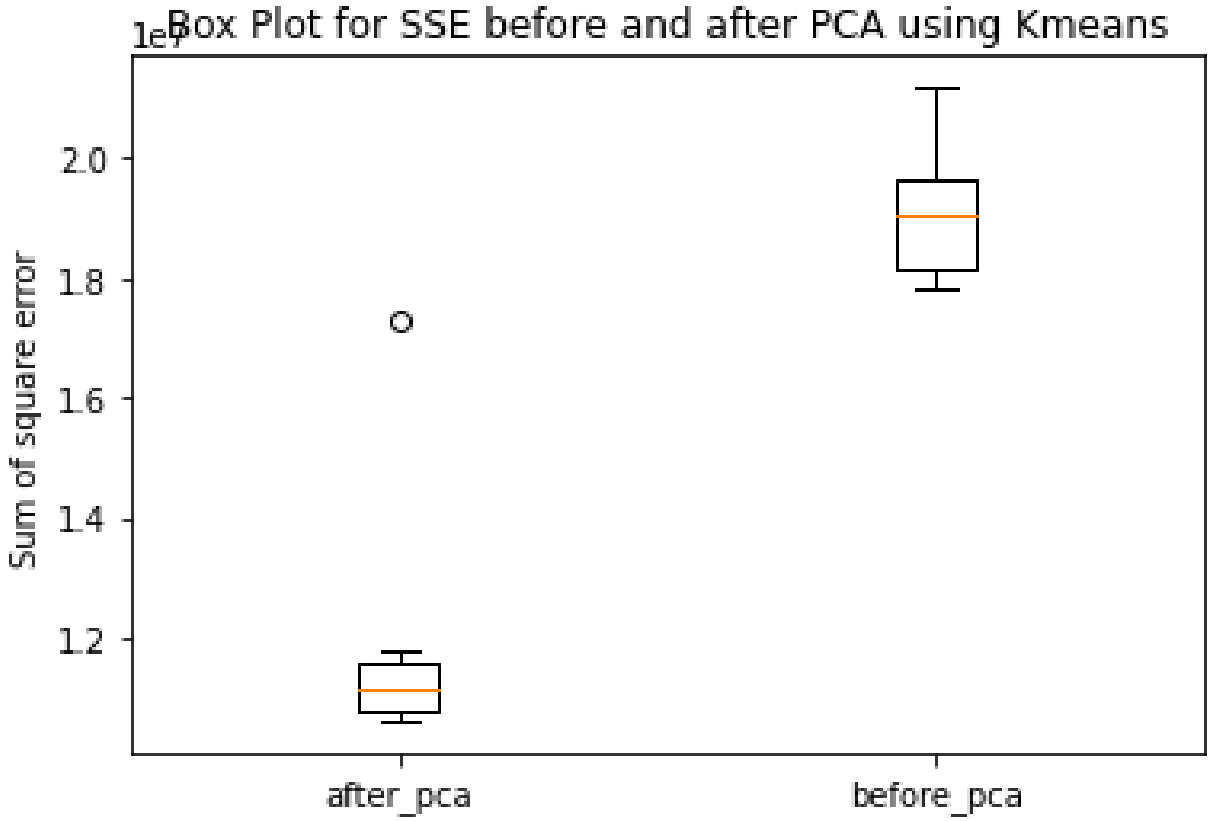
```
combined_df.loc['before_pca']['sse']])
plt.xticks([1, 2], ['after_pca', 'before_pca'])
plt.ylabel('Sum of square error')
plt.title('Box Plot for SSE before and after PCA using
Kmeans')
plt.show()
```

Discussion of Experiments

Answer here...
Have ran the Kmeans Llyod algoritm on the actual dataset.Also, have perfomed PCA on the given dataset to perform dimensionality reduction and then ran Kmeans clustering algorithm on the reduced dataset. Here the number of features dropped significantly to 372 principal components. So kmeans was ran over 800 * 372 datapoints. Have used the within sum of square error as the validation technique to understand the kmeans performance.
Here the target variable/class has 5 unique cancer types.Performed label encoding on it, it randomly assigns the number to class. Here the cluster number and the cluster number after kmeans doesnot match or we cannot use that to check error, hence used sum of square error. Have ran kmeans for 20 times where k=5 for actual dataset and reduced dataset(PCA).From the graph it can be seen that the error reduces greatly after performing dimensionality reduction.
The median error after PCA is much less than the median error before PCA.From the error metric and run time perspective, Yes clustering did improve after PCA. But there could be other unseen challenges , as the main information to cluster may be lost as we are considering 90percent of variance, so this depends on the usecase.

Plot/s

Place images here with suitable captions.



(5)

Problem 2

Randomly choose 50 points from the RNA-Seq data set (call this data set RNA-Seq$_{50}$) and perform hierarchical clustering. You are allowed to use R/Python packages for this question (Ignore the class variable while performing hierarchical clustering.) **[25 pt.]**

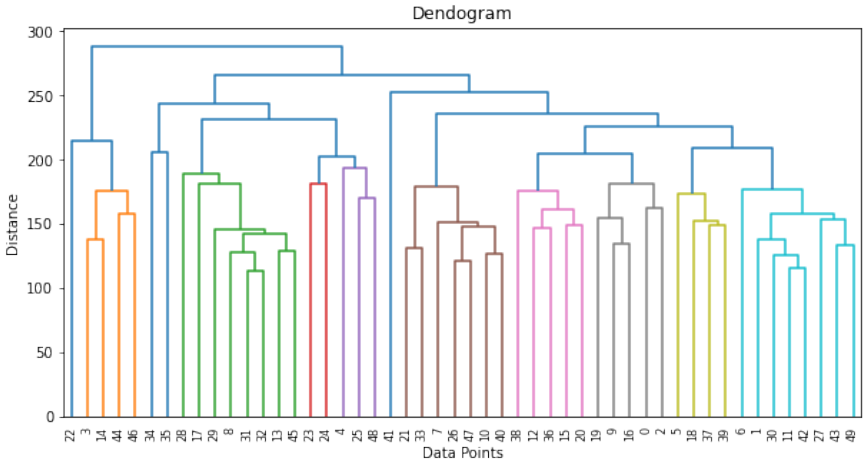
1. Using hierarchical clustering with complete linkage cluster RNA-Seq$_{50}$. Give the dendrogram.

R or Python script

```r
# Sample R Script With Highlighting
```

```python
# Sample Python Script With Highlighting

import pandas as pd
from sklearn.cluster import AgglomerativeClustering
from sklearn.preprocessing import StandardScaler
import numpy as np
```

```
import swifter
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import LabelEncoder
from tqdm import tqdm
import numpy as np
from sklearn.preprocessing import StandardScaler
from numpy import linalg as LA
import warnings
from scipy.spatial.distance import euclidean
from scipy.cluster.hierarchy import dendrogram



from scipy.cluster.hierarchy import linkage,
dendrogram,fcluster


df_data= pd.read_csv('TCGA-PANCAN-HiSeq-801x20531/data.csv')
df_data.drop(columns=['Unnamed: 0'],inplace=True)


def hierarchical_clustering(data):
    # Standardize the data to ensure all features have the same scale
    data_scaled = StandardScaler().fit_transform(data)

    # Perform hierarchical clustering with full linkage
    Z = linkage(data_scaled, method='complete')

    # Plot dendrogram
    plt.figure(figsize=(10,5))
    dendrogram(Z)
    plt.ylabel('Distance')
    plt.xlabel('Data Points')
    plt.title('Dendogram')
    plt.show()

    return Z,data_scaled

# Randomly select 50 data points from the dataset
df_50 = df_data.sample(n=50, random_state=42)
Z,data_scaled=hierarchical_clustering(df_50)
```



(6)

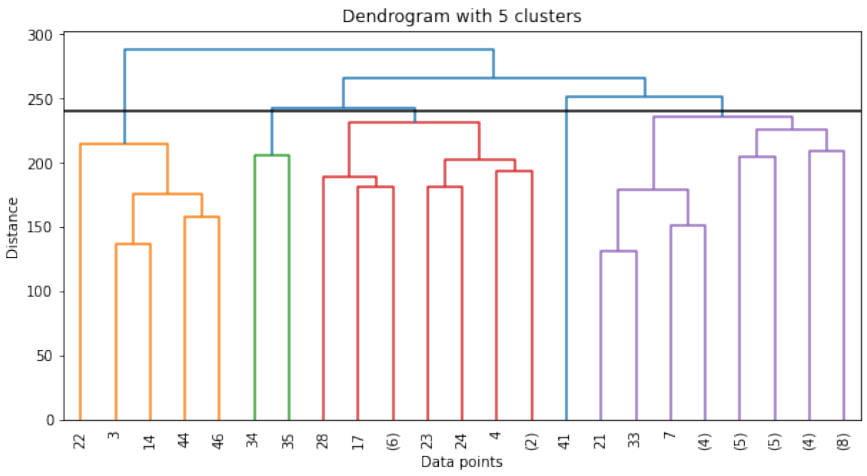
2. Cut the dendrogram at a height that results in 5 distinct clusters. Calculate the error-rate.

R or Python script

```r
# Sample R Script With Highlighting
```

```python
# Sample Python Script With Highlighting

def cut_dendogram(Z,k,max_d):
    distance_threshold = Z[-k, 2]
    clusters = fcluster(Z, distance_threshold, criterion='distance')

    # Print the number of clusters and their sizes
    n_clusters = len(np.unique(clusters))
    cluster_sizes = [np.sum(clusters == i) for i in range(1, n_clusters+1)]
    print('Number of clusters:{}'.format(n_clusters))
    print('Cluster sizes: {}'.format(cluster_sizes))

    # Plot dendrogram with cluster labels
    plt.figure(figsize=(10, 5))
    dendrogram(Z, truncate_mode='level', p=5, color_threshold=max_d)
    plt.title('Dendrogram with {} clusters'.format(n_clusters))
    plt.xlabel('Data points')
    plt.ylabel('Distance')
    plt.xticks(rotation=90)
    plt.axhline(y=max_d, c='k')
    plt.show()
    return clusters



clusters=cut_dendogram(Z,5,240)

def new_centroids(df_label, df1):
    df1['Label']=df_label
```

```
    # Calculate the mean of the rows with the same label
    return df1.groupby('Label').mean() # Transpose the
    dataframe to have the new centroids as columns and index as feature name


def sum_of_square_error(new_centroids, data, labels):
    """
    Computes the sum of squared errors between the data points and their assigned centroids.

    Args:
    new_centroids (DataFrame): The new centroids computed in the current iteration.
    data (DataFrame): The input data points.
    labels : The labels assigned to each data point.

    Returns:
    The sum of squared errors.
    """
    # Transpose the new centroids dataframe and reset the index
    new_centroids = new_centroids.reset_index()
    # Get the columns of the data dataframe
    columns = data.columns
    # Join the data dataframe and the labels clusters
    data['Label'] = labels
    sse = []
    # Compute the distance between each data point and its assigned centroid
    for i in range(1,len(new_centroids)):
        distance = np.sum(np.square(data[data['Label']==i]
        [columns] - new_centroids.iloc[i][columns]), axis=1)
        sse.append(sum(distance))
    # Return the sum of squared errors
    return sum(sse)


df_50_scaled= pd.DataFrame(data_scaled)
df_50_scaled.columns= df_data.columns

df_new_centroids = new_centroids(clusters, df_50_scaled)
sse=sum_of_square_error(df_new_centroids, df_50_scaled,
clusters)
print('The within cluster sum of square error rate is {}'.format(sse))

#
#The within cluster sum of square error rate is #2139434.5118071763
```



(7)

Discussion of Experiments

Answer here...

Have performed Hierarchical clustering on the actual datset where the Euclidean distance is used and the method is complete.
We exactly required 5 clusters to be formed. Have used distance threshold to form form the required number of clusters.Also, the dendogram is cut at a particular distance to visually understand the five unique clusters.
Have calculated the within sum of square error rate. Before PCA, the error rate of the actual data where method=complete and number of clusters=5 is 2139434.5

3. First, perform PCA on RNA-Seq$_{50}$ (Keep 90% of variance). Then hierarchically cluster the reduced data using complete linkage and Euclidean distance. Report the dendrogram.

R or Python script

```r
# Sample R Script With Highlighting
```

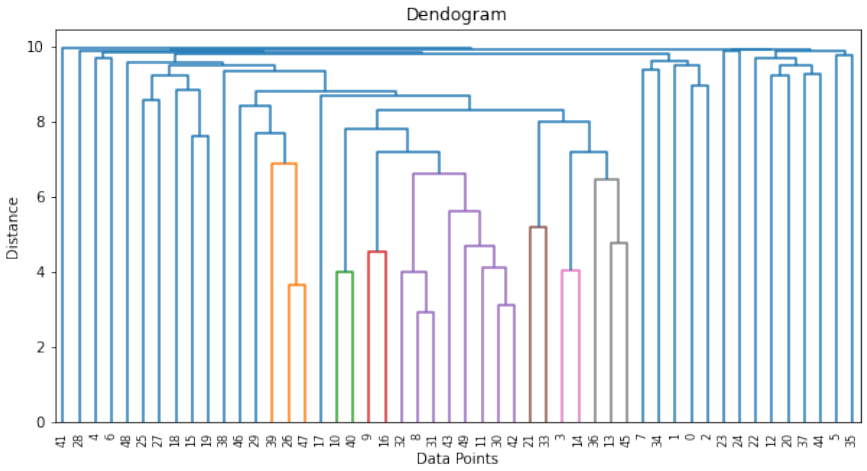
```python
# Sample Python Script With Highlighting


import pandas as pd
import swifter
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import LabelEncoder
from tqdm import tqdm
import numpy as np
from sklearn.preprocessing import StandardScaler
from numpy import linalg as LA
import warnings


```

```
def PCA(df,threshold):
    ## Performing Standardization so that all features are
    given same importance initially and each feature can contribute
    ## equally to PC irrespective of scale or magnitude
    df =pd.DataFrame(StandardScaler().fit_transform(df))
    ## Performing Data Centering on standardized dataframe
    centred_df= df-np.mean(df,axis=0)
    ## Calculating Covariance
    covariance=np.cov(df.T)
    eigen_values, eigen_vectors = LA.eig(covariance)
    ## Sorting the eigen values in descending order, using argsort, we get the indices of eigen
    sorted_index=eigen_values.argsort()[::-1]
    df_variance=pd.DataFrame(eigen_values[sorted_index]/sum(
    eigen_values),columns=['variance'])
    df_variance['cumulative_variance']= df_variance['variance'].cumsum()
    ## Number of principal components required to cover variance uptill certian threshold
    df_number_pc_var=df_variance[df_variance['cumulative_variance']<= threshold]
    number_of_pc=len(df_number_pc_var)
    print("The number of principal components required to cover {} percent variance are
    {}".format(threshold*100,number_of_pc))
    ## Selecting the required number of eigen vectores for performing dot product
    selected_eigen_vectors=eigen_vectors[:,sorted_index[:number_of_pc]]

    #Projecting data over the selected number of principal components
    principal_component=centred_df.dot(selected_eigen_vectors) #Performing dot product
    principal_component.columns=[f'PC{i+1}' for i in range(number_of_pc)]
    ## Creating Scree plots

    fig, axes = plt.subplots(1, 2, figsize=(20, 6))

    # Plot the first subplot of variance on the left side
    axes[0].plot(range(1,len(df_number_pc_var)+1),
    df_number_pc_var['variance'],'ro-', linewidth=2)
    axes[0].set_title('Scree Plot for Variance')
    axes[0].set_ylabel('Variance')
    axes[0].set_xlabel('Number of Principal Component')
    axes[0].set_xticks(np.arange(1,
    len(df_number_pc_var)+1, 1),rotation=90)

    # Plot the second subplot of cumulative variance on the right side
    axes[1].plot(range(1,len(df_number_pc_var)+1),
    df_number_pc_var['cumulative_variance'],'ro-', linewidth=2)
    axes[1].set_title('Scree Plot for Cumulative Variance')
    axes[1].set_ylabel('Cumulative Variance')
    axes[1].set_xlabel('Number of Principal Component')
    axes[1].set_xticks(np.arange(1,
    len(df_number_pc_var)+1, 1),rotation=90)
    plt.show()
    print('Correlation of PC1,PC2
    \n',principal_component[['PC1','PC2']].corr())
    plt.scatter(principal_component['PC1'],principal_compone
    nt['PC2'])
    plt.xlabel('PC1')
```

```
    plt.ylabel('PC2')
    plt.title('Scatter plot of PC1 VS PC2')
    plt.show()
    print('The Total variance explained by PC1,PC2 is {}
    percent'.format(round(np.real(df_number_pc_var['cumulati
    ve_variance'][1])*100,2)))

    loadings =
    pd.DataFrame(selected_eigen_vectors,index=df.columns)
    warnings.filterwarnings("ignore")

    return principal_component,loadings,df_variance
principal_component,loadings,df_variance=PCA(df_50,0.9)
df_kmeans1=
pd.DataFrame(np.real(principal_component.values), columns=principal_component.columns)
Z,data_scaled=hierarchical_clustering(df_kmeans1)
```

Plot/s

Place images here with suitable captions.



(8)

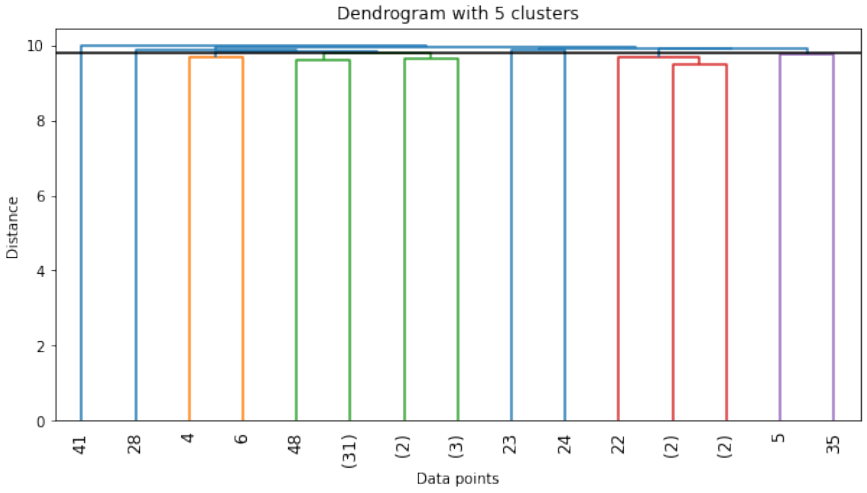
4. Cut the dendrogram at a height that results in 5 distinct clusters. Give the error-rate. Discuss your findings, i.e., how did PCA affect hierarchical clustering results?

R or Python script

```
# Sample R Script With Highlighting
```

```
# Sample Python Script With Highlighting
clusters=cut_dendogram(Z,5,9.8)


df_50_scaled= pd.DataFrame(df_kmeans1)
df_50_scaled.columns= df_kmeans1.columns
```

```
df_new_centroids = new_centroids(clusters, df_50_scaled)
sse=sum_of_square_error(df_new_centroids, df_50_scaled, clusters)
print('The within cluster sum of square error rate is {}'.format(sse))
#The within cluster sum of square error rate is #1611959.1275101865
```



(9)

Discussion of Experiments

Answer here...
Firstly, performed PCA over theactual dataset to generate a reduced dataset covering 90percent of variance.
34 Principal components are required for covering 90percent of variance. Have performed Hierarchical clustering on the reduced datset where the Euclidean distance is used and the method is complete. We exactly required 5 clusters to be formed. Have used distance threshold to form form the required number of clusters.
Also, the dendogram is cut at a particular distance to visually understand the five unique clusters. Have calculated the within sum of square error rate. After PCA, the error rate of the actual data where method=complete and number of clusters=5 is 1611959.1. Before applying PCA, the error rate was 2139434.5. Hence it can be seen that the error rate decreased greatly after applying PCA.
Principal Component Analysis (PCA) is a technique that can be utilized to decrease the number of variables used to calculate the distance between observations, which can in turn decrease the within sum of square errors when performing hierarchical clustering. If a dataset contains numerous variables that are highly correlated, including all of these variables in the distance calculation can result in redundant information and an overemphasis on certain features. However, using PCA to reduce the number of variables can help to focus on the most important features of the data and eliminate the effects of correlated variables. Consequently, this can lead to a more accurate clustering solution and a decrease in the within sum of square errors.

Problem 3

Run your k-means clustering program from previous assignment for 20 runs and hierarchical clustering with three different linkage techniques over the RNA-Seq data set. Compare those 4 different clustering algorithms

for $k = 5$ using appropriate cluster validity techniques, i.e., internal, external or relative indices. Plots are generally a good way to convey complex ideas quickly, i.e., box plots, whisker plots. Discuss your results, i.e., Did clustering improve after PCA? What algorithm performs best before and after PCA? **[25 pt.]**

R or Python script

```r
# Sample R Script With Highlighting
```

```python
# Sample Python Script With Highlighting
import pandas as pd
import swifter
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import LabelEncoder
from tqdm import tqdm
import numpy as np
from sklearn.preprocessing import StandardScaler
from numpy import linalg as LA
import warnings



df_data= pd.read_csv('TCGA-PANCAN-HiSeq-801x20531/data.csv')
df_label=pd.read_csv('TCGA-PANCAN-HiSeq-801x20531/labels.csv')

df_data.drop(columns=['Unnamed: 0'],inplace=True)
df_label.drop(columns=['Unnamed: 0'],inplace=True)

# Performing Kmeans clustering

import pandas as pd
import swifter
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import LabelEncoder
from tqdm import tqdm
import numpy as np
from sklearn.preprocessing import StandardScaler
from numpy import linalg as LA
import warnings
from scipy.spatial.distance import euclidean
from sklearn.metrics import silhouette_score, calinski_harabasz_score

import time
def initialize_centroids(df, k):
    """
    Function to initialize random centroids from dataset.
    Input:
        - df: pandas dataframe with the data
        - k: integer number of clusters
    Output:
        - temp_df: pandas dataframe with the centroids as columns and index as label
    """
```

```
    centroids = []
    for i in range(k):
        centroids.append(df.apply(lambda x: float(x.sample())))
        # Take a random sample from each column to create a centroid
    centroids = pd.concat(centroids, axis=1)
    centroids.index.name = 'Label'

    return centroids


def assign_labels(df, centroids):
    """
    Function to calculate the closest centroid label for each row in a dataframe.
    Input:
        - df: pandas dataframe with the data
        - centroids: pandas dataframe with the centroids as columns and index as label
    Output:
        - distances.idxmin(axis=1): pandas series with the label of the closest centroid for each rou
    """
    distances = centroids.swifter.apply(lambda x:
    np.sqrt(((df - x) ** 2).sum(axis=1)))
    # Calculate the Euclidean distance between each row in df and each centroid
    return distances.idxmin(axis=1)
    # Get the index of the minimum distance, which corresponds to the label of the closest centroid


def new_centroids(df_label, df1):
    """
    Function to calculate the new centroids based on the current labels of the rows.
    Input:
        - df_label: pandas series with the label of the closest centroid for each row in df1
        - df1: pandas dataframe with the data
    Output:
        - new_centroids.T: pandas dataframe with the new centroids as columns and index as feature na
    """
    joined_df = df1.join(df_label)
    joined_df.rename(columns={0: 'Label'}, inplace=True)
    # Rename the column with the label
    # Calculate the mean of the rows with the same label
    return joined_df.groupby('Label').mean().T
    # Transpose the dataframe to have the new centroids as columns and index as feature name


def sum_of_square_error(new_centroids, data, labels):
    """
    Computes the sum of squared errors between the data points and their assigned centroids.

    Args:
    new_centroids (DataFrame): The new centroids computed in the current iteration.
    data (DataFrame): The input data points.
    labels (DataFrame): The labels assigned to each data point.

    Returns:
```

```
    The sum of squared errors.
    """
    # Transpose the new centroids dataframe and reset the index
    new_centroids = new_centroids.T.reset_index()
    # Get the columns of the data dataframe
    columns = data.columns
    # Join the data dataframe and the labels dataframe
    data = data.join(labels)
    # Rename the '0' column of the labels dataframe to 'Label'
    data.rename(columns={0:'Label'}, inplace=True)
    sse = []
    # Compute the distance between each data point and its assigned centroid
    for i in range(len(new_centroids)):
        distance = np.sum(np.square(data[data['Label']==i]
        [columns] - new_centroids.iloc[i][columns]), axis=1)
        sse.append(sum(distance))
    # Return the sum of squared errors
    return sum(sse)


#Calculating silhouette coefficient
def silhouette_coef(df_data,clusters):
    silhouette_avg = silhouette_score(df_data, clusters)
    return silhouette_avg

# Calculating Calinski-Harabasz index
def Calinski_index(df_data,clusters):
    calinski_i = calinski_harabasz_score(df_data, clusters)
    return calinski_i


def kmeans_lyod_with_error(df1, k, tou):
    """
    Function to run the K-means Lloyd algorithm.
    Input:
        - df1: pandas dataframe with the data
        - k: integer number of clusters
        - tou: float tolerance level to stop the algorithm
    Output:
        - centroids: pandas dataframe with the final centroids as columns and index as label
    """
    start_time=time.time()
    centroids = initialize_centroids(df1, k) # Initialize random centroids
    initial_list_of_columns = centroids.columns.to_list()
    iteration = 0
    while True:
        # Assign labels to current centroids
        df_label = assign_labels(df1, centroids)
        df_label = pd.DataFrame(df_label)
        # Calculate new centroids
        df_new_centroids = new_centroids(df_label, df1)
        new_list_of_columns = df_new_centroids.columns.to_list()
        # Keep the number of clusters the same i.e maintain same k
```

```
        for i in initial_list_of_columns:
            if i not in new_list_of_columns:
                df_new_centroids[i] = centroids[i]
        # Calculate tao
        distance = []
        for col in centroids.columns:
            col_distance = euclidean(centroids[col], df_new_centroids[col])
            distance.append(col_distance)
        tao_calculated=sum(distance)/k #Used the formula provided for calculating Tao

        sse = sum_of_square_error(df_new_centroids, df1, df_label)
        silhouette=silhouette_coef(df_data,df_label[0].toli
        st())
        calinski=
        Calinski_index(df_data,df_label[0].tolist())
        #error=error_clusters(df_label,df1,k)
        end_time= time.time()
        if iteration>100:
            print("Iteration exceeded")

            return  sse,end_time-start_time,silhouette,calinski
            break

        if tao_calculated<tou or iteration >100:   #if the
        convergence is met, kmeans will stop  or else if
        the convergence is never met, after 100 iteration code will stop
            return  sse,end_time-
            start_time,silhouette,calinski
            break                                   #otherwise indefinite loop
        else:
            centroids= df_new_centroids
            # In case we need more iterations, the
            centroids calculated at this step acts as input
        iteration+=1


#Kmeans Clustering before PCA, on raw data
error_matrix=[]
for i in range (1,21):
    sse,run_time,silhouette,calinski=kmeans_lyod_with_error(df_data,5,10)
    error_matrix.append([i,sse,run_time,silhouette,calinski])
error_matrix_df_kmeans_before_pca=
pd.DataFrame(error_matrix,columns=[ 'repetition','sse','run_time','silhouette','calinski'])
error_matrix_df_kmeans_before_pca



# Performing PCA on dataset
def PCA(df,threshold):
    ## Performing Standardization so that all features are
    given same importance initially and each feature can contribute
    ## equally to PC irrespective of scale or magnitude
    df =pd.DataFrame(StandardScaler().fit_transform(df))
```

```
    ## Performing Data Centering on standardized dataframe
    centred_df= df-np.mean(df,axis=0)
    ## Calculating Covariance
    covariance=np.cov(df.T)
    eigen_values, eigen_vectors = LA.eig(covariance)
    ## Sorting the eigen values in descending order, using
    argsort, we get the indices of eigen values in descending order
    sorted_index=eigen_values.argsort()[::-1]
    df_variance=pd.DataFrame(eigen_values[sorted_index]/sum
    (eigen_values),columns=['variance'])
    df_variance['cumulative_variance']= df_variance['variance'].cumsum()
    ## Number of principal components required to cover variance uptill certian threshold
    df_number_pc_var=df_variance[df_variance['cumulative_va
    riance']<= threshold]
    number_of_pc=len(df_number_pc_var)
    print("The number of principal components required to
    cover {} percent variance are {}".format(threshold*100,number_of_pc))
    ## Selecting the required number of eigen vectores for performing dot product
    selected_eigen_vectors=eigen_vectors[:,sorted_index[:number_of_pc]]

    #Projecting data over the selected number of principal components
    principal_component=centred_df.dot(selected_eigen_vecto
    rs) #Performing dot product
    principal_component.columns=[f'PC{i+1}' for i in range(number_of_pc)]
    ## Creating Scree plots

    fig, axes = plt.subplots(1, 2, figsize=(20, 6))

    # Plot the first subplot of variance on the left side
    axes[0].plot(range(1,len(df_number_pc_var)+1),
    df_number_pc_var['variance'],'ro-', linewidth=2)
    axes[0].set_title('Scree Plot for Variance')
    axes[0].set_ylabel('Variance')
    axes[0].set_xlabel('Number of Principal Component')
    axes[0].set_xticks(np.arange(1,
    len(df_number_pc_var)+1, 1))

    # Plot the second subplot of cumulative variance on the right side
    axes[1].plot(range(1,len(df_number_pc_var)+1),
    df_number_pc_var['cumulative_variance'],'ro-', linewidth=2)
    axes[1].set_title('Scree Plot for Cumulative Variance')
    axes[1].set_ylabel('Cumulative Variance')
    axes[1].set_xlabel('Number of Principal Component')
    axes[1].set_xticks(np.arange(1, len(df_number_pc_var)+1, 1))
    plt.show()
    print('Correlation of PC1,PC2 \n',principal_component[['PC1','PC2']].corr())

    plt.scatter(principal_component['PC1'],principal_compon
    ent['PC2'])
    plt.xlabel('PC1')
    plt.ylabel('PC2')
    plt.title('Scatter plot of PC1 VS PC2')
    plt.show()
```

```
    print('The Total variance explained by PC1,PC2 is {}
    percent'.format(round(np.real(df_number_pc_var['cumulat
    ive_variance'][1])*100,2)))

    loadings =
    pd.DataFrame(selected_eigen_vectors,index=df.columns)
    warnings.filterwarnings("ignore")

    return
    principal_component,loadings,df_variance,df_number_pc_v
    ar,eigen_values, eigen_vectors

principal_component,loadings,df_variance,df_number_pc_var,e
igen_values, eigen_vectors=PCA(df_data,0.99)


# K means Clustering after PCA
error_matrix=[]
principal_component_r=
pd.DataFrame(np.real(principal_component.values), columns=principal_component.columns)
for i in range (1,21):
    sse,run_time,silhouette,calinski=kmeans_lyod_with_error
    (principal_component_r,5,10)
    error_matrix.append([i,sse,run_time,silhouette,calinski
    ])
error_matrix_df_kmeans_after_pca= pd.DataFrame(error_matrix,columns=[
'repetition','sse','run_time','silhouette','calinski'])
error_matrix_df_kmeans_after_pca


# Plots for kmeans before and after PCA

plt.boxplot([combined_df_kmeans.loc['after_pca']['sse'],
combined_df_kmeans.loc['before_pca']['sse']])
plt.xticks([1, 2], ['after_pca', 'before_pca'])
plt.ylabel('Sum of square error')
plt.title('Box Plot for SSE before and after PCA using Kmeans')
plt.show()


plt.boxplot([combined_df_kmeans.loc['after_pca']
['silhouette'], combined_df_kmeans.loc['before_pca']
['silhouette']])
plt.xticks([1, 2], ['after_pca', 'before_pca'])
plt.ylabel('silhouette coefficient')
plt.title('Box Plot for silhouette coefficient before and after PCA using Kmeans')
plt.show()


plt.boxplot([combined_df_kmeans.loc['after_pca']
['calinski']], combined_df_kmeans.loc['before_pca']
['calinski']])
plt.xticks([1, 2], ['after_pca', 'before_pca'])
```

```
plt.ylabel('calinski index')
plt.title('Box Plot for calinski index before and after
PCA using Kmeans')
plt.show()


# Performing hierarchical clustering

import pandas as pd
from sklearn.cluster import AgglomerativeClustering
from sklearn.preprocessing import StandardScaler
import numpy as np

import swifter
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import LabelEncoder
from tqdm import tqdm
import numpy as np
from sklearn.preprocessing import StandardScaler
from numpy import linalg as LA
import warnings
from scipy.spatial.distance import euclidean
from scipy.cluster.hierarchy import dendrogram
from sklearn.metrics import silhouette_score, calinski_harabasz_score


from scipy.cluster.hierarchy import linkage,
dendrogram,fcluster



def hierarchical_clustering(data,method,k):
    # Standardize the data to ensure all features have the same scale
    data_scaled = StandardScaler().fit_transform(data)

    # Perform hierarchical clustering with full linkage
    Z = linkage(data_scaled, method=method)

    distance_threshold = Z[-k, 2]
    clusters = fcluster(Z, distance_threshold, criterion='distance')

    # Print the number of clusters and their sizes
    n_clusters = len(np.unique(clusters))
    cluster_sizes = [np.sum(clusters == i) for i in range(1, n_clusters+1)]
    print('Number of clusters:{}'.format(n_clusters))
    print('Cluster sizes: {}'.format(cluster_sizes))
    return clusters,data_scaled



def new_centroids(df_label, df1):
    df1['Label']=df_label
```

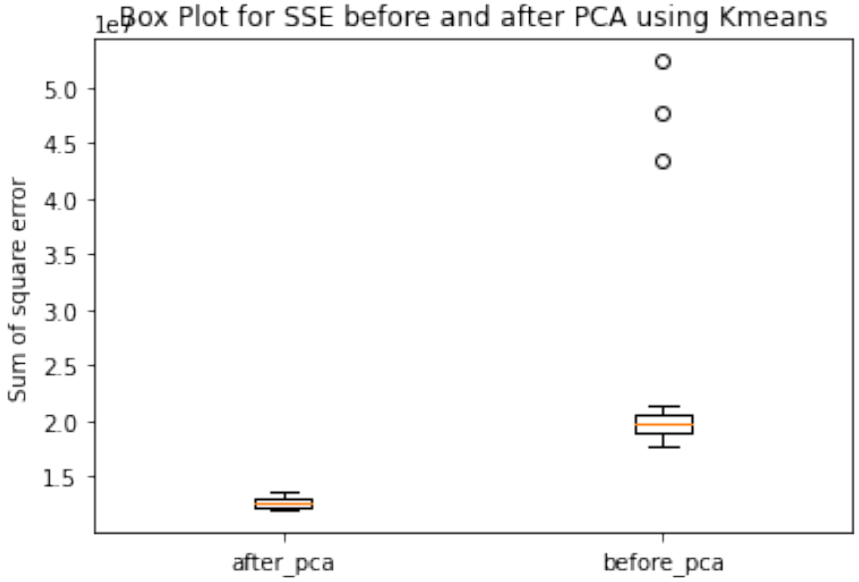
```
    # Calculate the mean of the rows with the same label
    return df1.groupby('Label').mean() # Transpose the
    dataframe to have the new centroids as columns and
    index as feature name


def sum_of_square_error(new_centroids, data, labels):
    """
    Computes the sum of squared errors between the data points and their assigned centroids.

    Args:
    new_centroids (DataFrame): The new centroids computed in the current iteration.
    data (DataFrame): The input data points.
    labels : The labels assigned to each data point.

    Returns:
    The sum of squared errors.
    """
    # Transpose the new centroids dataframe and reset the index
    new_centroids = new_centroids.reset_index()
    # Get the columns of the data dataframe
    columns = data.columns
    # Join the data dataframe and the labels clusters
    data['Label'] = labels
    sse = []
    # Compute the distance between each data point and its assigned centroid
    for i in range(1,len(new_centroids)):
        distance = np.sum(np.square(data[data['Label']==i]
        [columns] - new_centroids.iloc[i][columns]), axis=1)
        sse.append(sum(distance))
    # Return the sum of squared errors
    return sum(sse)

#Calculating silhouette coefficient
def silhouette_coef(df_data,clusters):
    silhouette_avg = silhouette_score(df_data, clusters)
    return silhouette_avg

# Calculating Calinski-Harabasz index
def Calinski_index(df_data,clusters):
    ch_score = calinski_harabasz_score(df_data, clusters)
    return ch_score

def clustering_runs(df_data,method,k):
    # Call clustering to return the clusters assigned
    after performing the clustering based on method
    clusters,data_scaled=hierarchical_clustering(df_data,me
    thod,k)
    # Scaling the data ti calculate the within sum of square error,  silhouette_coef, Calinski_index
    df_scaled= pd.DataFrame(data_scaled)
    df_scaled.columns= df_data.columns
    df_new_centroids = new_centroids(clusters, df_scaled)
    sse=sum_of_square_error(df_new_centroids, df_scaled, clusters)
```

```
    silhouette=silhouette_coef(df_scaled,clusters)
    Calinski=Calinski_index(df_scaled,clusters)
    return sse,silhouette,Calinski


# Hierarchical clustering after PCA
method_list=['single','average','complete','ward']
principal_component_r=
pd.DataFrame(np.real(principal_component.values), columns=principal_component.columns)
clustering_error=[]
for i in method_list:
    sse,silhouette,Calinski=clustering_runs(principal_component_r,i,5)
    clustering_error.append([i,sse,silhouette,Calinski])
clustering_error_df_afterpca=
pd.DataFrame(clustering_error,columns=[
'method','within_sse','silhouette','Calinski'])
print(clustering_error_df_afterpca)


# Hierarchical clustering before PCA, on raw data
method_list=['single','average','complete','ward']
clustering_error=[]
df_data= pd.read_csv('TCGA-PANCAN-HiSeq-801x20531/data.csv')
df_data.drop(columns=['Unnamed: 0'],inplace=True)
for i in method_list:
    sse,silhouette,Calinski=clustering_runs(df_data,i,5)
    clustering_error.append([i,sse,silhouette,Calinski])
clustering_error_df_beforepca=
pd.DataFrame(clustering_error,columns=[
'method','within_sse','silhouette','Calinski'])
print(clustering_error_df_beforepca)


# Plots for Hierarchical clustering before and after PCA
merged_df_cluster =
pd.merge(clustering_error_df_beforepca,
clustering_error_df_afterpca, on='method', suffixes=
('_before_pca', '_after_pca'))

merged_df_cluster[['method','within_sse_before_pca','within
_sse_after_pca']].plot(x='method', kind='bar')
plt.ylabel('Within Sum of Square Error')
plt.title('Box Plot for Within Sum of Square Error before
and after PCA ')
plt.show()

merged_df_cluster[['method','silhouette_before_pca','silhouette_after_pca']].plot(x='method', kind='b
plt.ylabel('Silhouette coefficient')
plt.title('Bar Plot for Silhouette coefficient before and after PCA ')
plt.show()

merged_df_cluster[['method','Calinski_before_pca','Calinski
```

```
_after_pca']].plot(x='method', kind='bar')
plt.ylabel('Calinski Index')
plt.title('Bar Plot for Calinski Index before and after
PCA ')
plt.show()

after=pd.DataFrame(error_matrix_df_kmeans_after_pca.mean()).T
before=pd.DataFrame(error_matrix_df_kmeans_before_pca.mean()).T
after.drop(columns=['repetition'],inplace=True)
before.drop(columns=['repetition'],inplace=True)
after['method']='kmeans'
before['method']='kmeans'
merge = pd.merge(after, before,on='method', suffixes=
('_before_pca', '_after_pca'))
merge.drop(columns=
['run_time_before_pca','run_time_after_pca'],inplace=True)
merge.columns=[ 'within_sse_after_pca',
'silhouette_after_pca',
       'Calinski_after_pca',
       'method','within_sse_before_pca',
       'silhouette_before_pca',
       'Calinski_before_pca']
df_concat = pd.concat([merged_df_cluster, merge])

df_concat[['method','within_sse_before_pca','within_sse_aft
er_pca']].plot(x='method', kind='bar')
plt.ylabel('Within Sum of Square Error')
plt.title('Box Plot for Within Sum of Square Error before
and after PCA using all clustering methods')
plt.show()

df_concat[['method','silhouette_before_pca','silhouette_aft
er_pca']].plot(x='method', kind='bar')
plt.ylabel('Silhouette coefficient')
plt.title('Bar Plot for Silhouette coefficient before and
after PCA using all clustering methods')
plt.show()

df_concat[['method','Calinski_before_pca','Calinski_after_p
ca']].plot(x='method', kind='bar')
plt.ylabel('Calinski Index')
plt.title('Bar Plot for Calinski Index before and after
PCA using all clustering methods')
plt.show()
```

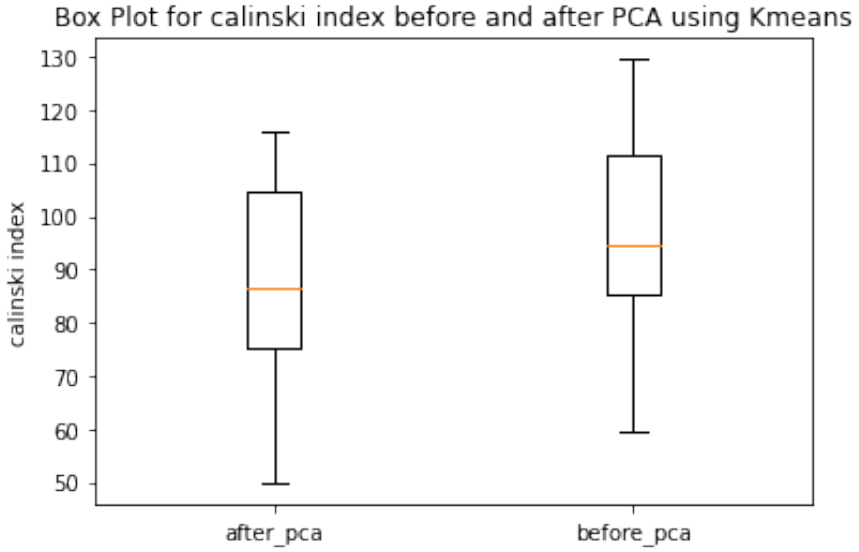
Plot/s

Place images here with suitable captions.

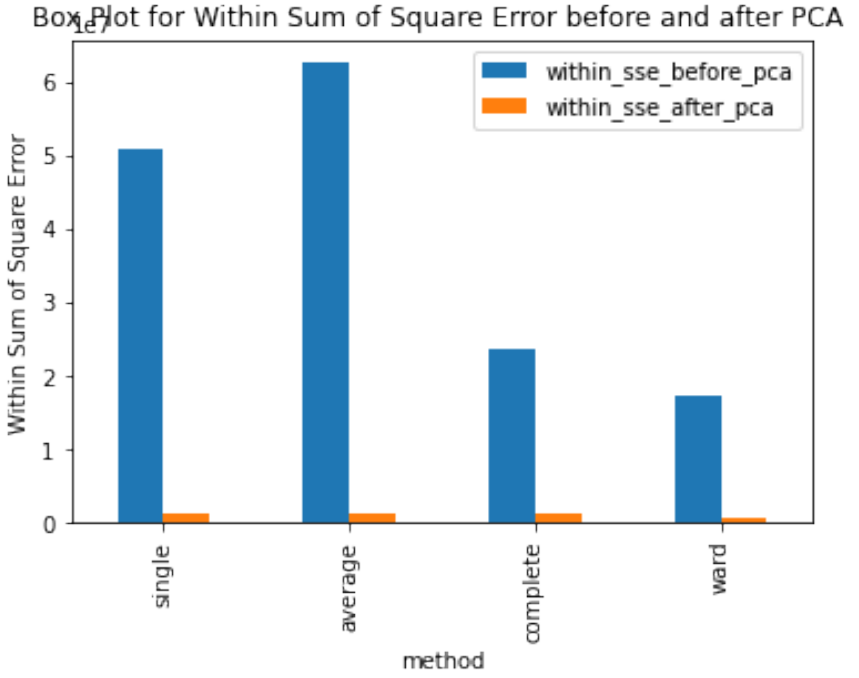


(10)

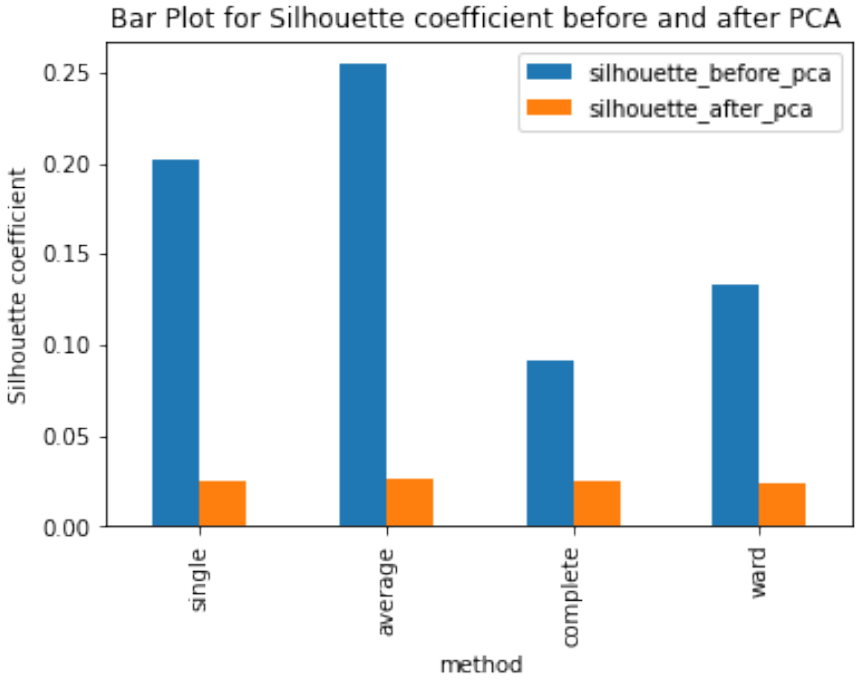
(11)


Box Plot for calinski index before and after PCA using Kmeans
calinski index
130
120
110
100
90
80
70
60
50
after_pca
before_pca

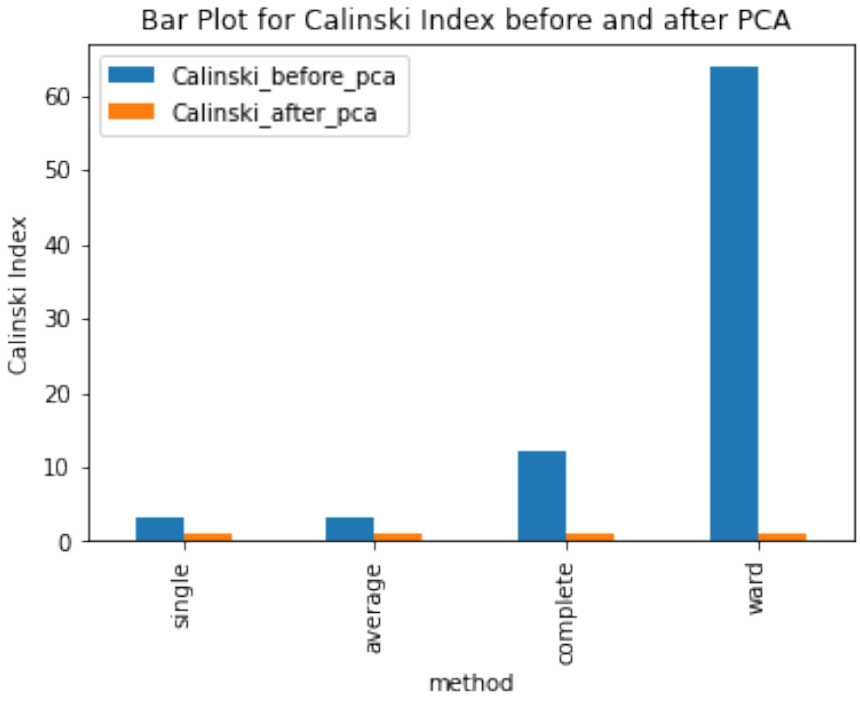
(12)


Box Plot for Within Sum of Square Error before and after PCA
1e7
Within Sum of Square Error
6
5
4
3
2
1
0
within_sse_before_pca
within_sse_after_pca
single
average
complete
ward
method

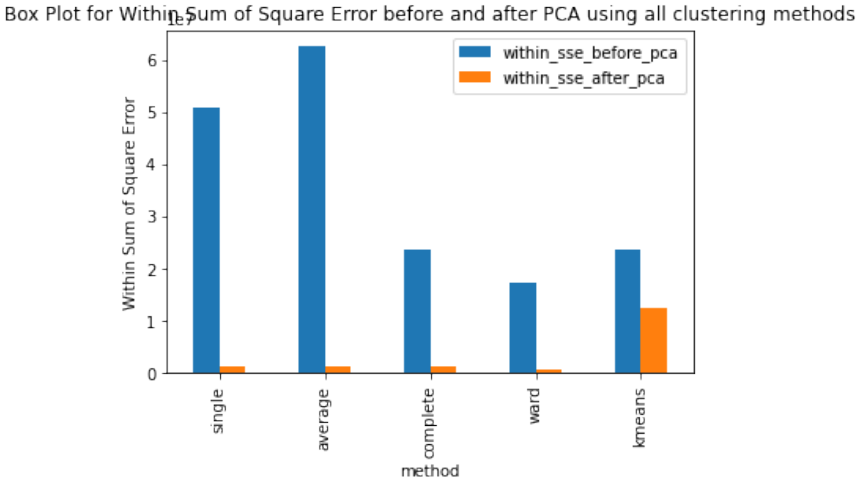
(13)


Bar Plot for Silhouette coefficient before and after PCA
silhouette_before_pca
silhouette_after_pca
Silhouette coefficient
0.25
0.20
0.15
0.10
0.05
0.00
single
average
complete
ward
method

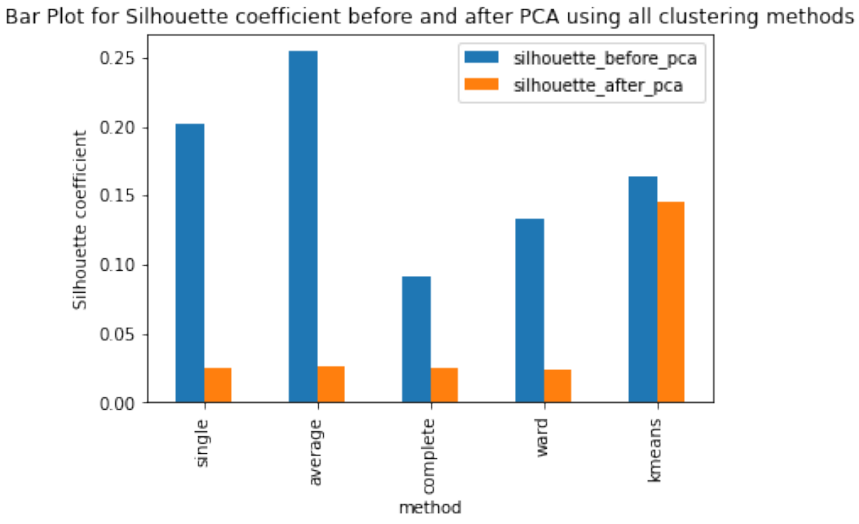
(14)


Bar Plot for Calinski Index before and after PCA
Calinski_before_pca
Calinski_after_pca
Calinski Index
60
50
40
30
20
10
0
single
average
complete
ward
method

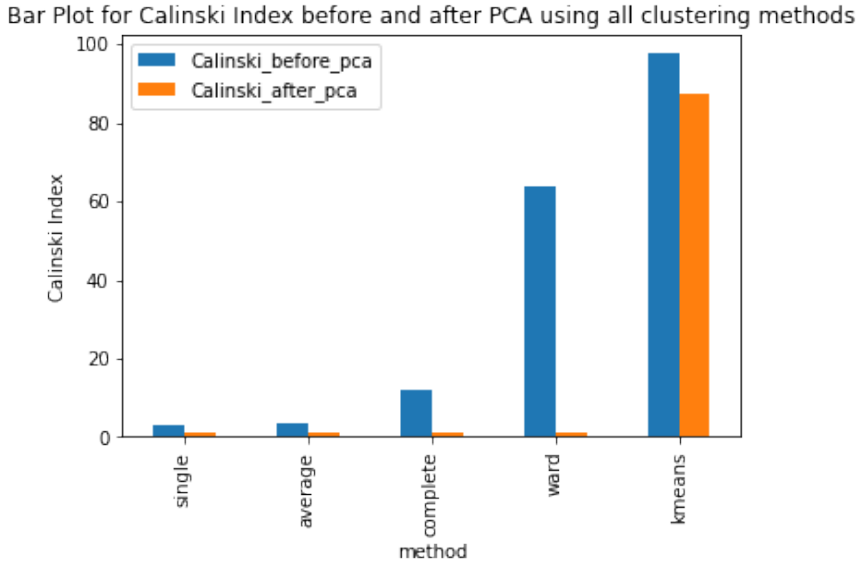
(15)


Box Plot for Within Sum of Square Error before and after PCA using all clustering methods
1e7
Within Sum of Square Error
within_sse_before_pca
within_sse_after_pca
single
average
complete
ward
kmeans
method

(16)


Bar Plot for Silhouette coefficient before and after PCA using all clustering methods
Silhouette coefficient
silhouette_before_pca
silhouette_after_pca
single
average
complete
ward
kmeans
method

(17)



(18)

Discussion of Experiments

Answer here...
Run your k-means clustering program from previous assignment for 20 runs and hierarchical clustering with three different linkage techniques over the RNA-Seq data set. Compare those 4 different clustering algorithms for k = 5 using appropriate cluster validity techniques, i.e., internal, external or relative indices. Plots are generally a good way to convey complex ideas quickly, i.e., box plots, whisker plots. Discuss your results, i.e., Did clustering improve after PCA? What algorithm performs best before and after PCA?
Have ran the kmeans clustering algorithm on the actual dataset for k=5 i.e number of clusters=5. Also ran 3 different hierarchical clustering algorithms on the actual dataset where the method for heirarchical clustering includes: 'single','average','complete','ward'. These methods are used to determine the distance between clusters based on the distances between their individual members.
Single linkage: calculates the distance between two clusters based on the shortest distance between any two points in the clusters.
Average linkage: calculates the distance between two clusters based on the average distance between all pairs of points in the clusters.
Complete linkage: calculates the distance between two clusters based on the maximum distance between any two points in the clusters.
Ward linkage: calculates the distance between two clusters based on the increase in the sum of squares that results when the two clusters are merged.
Each of these methods has its own strengths and weaknesses and the choice of method will depend on the specific characteristics of the data being analyzed and the objectives of the analysis. For example, single linkage is sensitive to noise and outliers, while complete linkage can produce clusters of uneven sizes. Ward linkage tends to produce compact, spherical clusters and is often used when the goal is to minimize the variance within clusters.
The different cluster validity techniques that are performed includes: The silhouette coefficient , Calinski-Harabasz index, and within sum of squares (WSS) are all appropriate techniques for evaluating the quality of clustering results. These indices are classified as internal ,relative and external indices.The silhouette index measures how well each data point fits into its assigned cluster compared to other clusters, the Calinski-Harabasz index measures the ratio of the between-cluster variance to the within-cluster variance, and the

WSS measures the sum of squared distances between each data point and the centroid of its assigned cluster. These indices can be used together to get a comprehensive evaluation of clustering results.
Have applied these validation techniques on the clusters formed after performing kmeans clustering and hierarchical clustering on the raw datset. The results of which are stored in dataframe. Perfomed PCA over the dataset to capture 99.99 variance, 724 principal components were generated. Applied kmeans clustering and hierarchical clustering on the reduced dataset (PCA) dataset and stored the results. From the graphs below it can be seen that: For kmeans clustering: The median within sum of square error after PCA is much less than that of the median within sum of square error after PCA. The median silhouette coefficient and Calinski index is also less for the data after PCA as compared to that of data before applying PCA (raw datset)
For hierarchical clustering: The bar length of within sum of square error before performing PCA is much higher than the bar length of within sum of square error after applying PCA for all types of clustering. Also, the silhouette coefficient and Calinski index shows similar results of after and before PCA.
Comparing all clustering algorithms:
It can be seen from the graph of within sum of square error that the bar length of average method of hierarchical clustering is the highest before PCA and that of ward is the smallest. So ward method performed well before applying PCA. After applying PCA, the ward has the smallest error as compared to kmeans and other hierarchical clustering methods.So for the given dataset, Ward can be used as the technique to perform clustering, considering within sum of square error as metric.
The silhouette coefficient for complete method is minimum when compared with that of other methods for both the dataset (before and after PCA).
The calinski index is maximum for Kmeans clustering and miminum for clustering using single linkage method.

Based on different validation techniques:
Ward performed best when the within sum of square error validation was used. Complete linkage outperformed when silhouette coefficient was used. Similarly, kmeans performed well using calinski index. Hence, it depends upon usecase and the given problem statement, to select a particular validation technique. Each has its pros and cons. To summarise, Clustering did improve after performing the PCA. THe error rates reduced.

Problem 4

The leader algorithm [1] represents each cluster using a point, known as a *leader*, and assigns each point to the cluster corresponding to the closest leader, unless this distance is above a user-specified threshold. In that case, the point becomes the leader in a new cluster **[10 pt.]**.
Hartigan's leader algorithm, proposed by John A. Hartigan in 1975, is a divisive approach to hierarchical clustering. It starts with all data points in a single cluster and splits them into smaller clusters recursively until a stopping criterion is met.
To begin the algorithm, a leader point is selected at random from the data set, which serves as the initial center of the first cluster. The distance between each point and the leader is then calculated, and each point is assigned to the closest cluster.
After assigning points to clusters, a new leader for each cluster is selected. This leader is the point with the smallest sum of distances to all the other points in the cluster. The process of calculating distances, assigning points to clusters, and selecting new leaders is repeated until a stopping criterion is met, such as a pre- defined number of clusters or a threshold for the distance between leaders of two clusters.

(a) What are the advantages and disadvantages of the leader algorithm as compared to K-means?

The advantages of Hartigan's leader algorithm over K-means algorithm can be summarized as follows:

No Voronoi diagram: Additionally, K-means requires an additional computational step of Voronoi diagram, which is not required for Hartigan's leader algorithm.

Reduction of distance calculations: In K-means clustering, distance calculations are performed for each data point with respect to all the cluster centers at every iteration. This can be computationally expensive, especially for large datasets or when the number of clusters is high. In contrast, Hartigan's leader algorithm reduces the number of distance calculations required by only comparing each data point to a single leader point, rather than all the cluster centers. This reduces the computational complexity of the algorithm.

It is important to note that K-means may be better suited for complex clusters, and is generally simpler to implement than Hartigan's algorithm. However, Hartigan's algorithm may be more effective in reducing the computational complexity of the clustering process.

Disadvantages of Hartingan's Algorithm over K-means:

Hartingan's Algorithm is sensitive to the first leader selection.

K-means is better suited for complex clusters.

K-means is a simpler algorithm and is generally used in real-life datasets with optimization parameters.

(b) Suggest ways in which the leader algorithm might be improved.
Leader's Algorithm can be improved in several ways.

One way to improve the algorithm is by selecting a better initial leader data point, as the algorithm is extremely sensitive to the initial leader point. A heuristic similar to K means++ initialization can be used.

Another improvement is through adaptive selection of the number of leader points. This can be done by adjusting the number of leader points during the clustering process to improve the algorithm's robustness and flexibility.

Parallelization can also be used to make distance calculations more efficient. Since we calculate the distance from the leader to every other point, this step can be parallelized to improve computational efficiency.

Problem 5

Clusters of documents can be summarized by finding the top terms (words) for the documents in the cluster, e.g., by taking the most frequent k terms, where k is a constant, say 10, or by taking all terms that occur more frequently than a specified threshold. Suppose that K-means is used to find clusters of both documents and words for a document data set **[10 pt.]**

(a) How might a set of term clusters defined by the top terms in a documents cluster differ from the word clusters found by clustering the terms with K-means?
The set of term clusters defined by the top terms in a documents cluster can differ from the word

clusters found by clustering the terms with K-means due to the different goals and levels of abstraction of each approach.

The set of term clusters defined by the top terms in a documents cluster focuses on summarizing the content of a group of related documents by identifying the most important and frequent terms in the cluster. In contrast, K-means clustering of terms seeks to find groups of terms that have similar patterns of co-occurrence across the entire document corpus, regardless of whether they appear frequently or infrequently in any one document.

Therefore, while the former approach is more focused on the specific content of a given set of documents, the latter approach captures more general patterns of term co- occurrence across the entire corpus.

Example

Let's say we have a collection of news articles about different sports. We want to group them into clusters based on the topics they cover. We could first extract the most frequent terms in each article and group the articles based on those terms. For example, one cluster might be defined by the terms "basketball," "LeBron James," and "NBA championship," indicating that the articles in that cluster are all related to the NBA playoffs.

Alternatively, we could cluster the terms themselves using K-means. This would group together terms that frequently appear together across the entire corpus, regardless of which articles they appear in. For example, we might find that terms like "basketball," "court," "team," and "score" all cluster together, indicating that they frequently co-occur in the sports articles.

In this example, the former approach (term clusters defined by the top terms in a document cluster) is more focused on the specific content of each article and the topics they cover, while the latter approach (word clusters found by clustering terms with K-means) captures more general patterns of term co-occurrence across the entire corpus.

(b) How could term clustering be used to define clusters of documents?
Term clustering is a useful method to define clusters of documents that share similar vocabulary. To achieve this, natural language processing (NLP) techniques such as creating a co-occurrence matrix of terms in the corpus can be used. Count Vectorizer can be employed to calculate the frequency of terms.

The co-occurrence matrix is a representation of the frequency of term co-occurrences across all documents in the corpus. Subsequently, a clustering algorithm such as K- means or hierarchical clustering can be applied to group frequently co-occurring terms into clusters.

Once the term clusters are defined, each document in the corpus can be represented as a vector of weights that indicates the frequency of terms in each cluster within the document. These document vectors can be used as input for another clustering algorithm to group similar documents together based on the similarity of their term cluster weights. Therefore, two rounds of clustering are performed: one to group frequently occurring words and another to group those frequently occurring words together to form clusters of similar documents.

Submission

You must use LaTeX to turn in your assignments. Please submit the following two files via Canvas:

1. A .pdf with the name `yourname-hw5-everything.pdf` which you will get after compiling your .tex file.

2. A .zip file with the name `yourname-hw5.zip` which should contain your .tex, .pdf, codes(.py, .ipynb, .R, or .Rmd), and a README file. The README file should contain information about dependencies and how to run your codes.

References

[1] J. Hartigan. *Clustering Algorithms.* John Wiley and Sons, New York, 1975.