B565-Data Mining
Homework #6

Due on Friday, March 31, 2023, 08:00 p.m.

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

Prem Amal

March 31, 2023

Expectation-Maximization Algorithm

This part is provided to help you implement the expectation-maximization algorithm.

```
1: function EXPECTATION-MAXIMIZATION($\mathbf{D}$, $k$, $\epsilon$)
2:     $t \leftarrow 0$
3:     Randomly initialize $\boldsymbol{\mu}_1^t, ..., \boldsymbol{\mu}_k^t$                          ▷ Initialization
4:     $\boldsymbol{\Sigma}_i^t \leftarrow \mathbf{I}, \forall i = 1, ..., k$
5:     $P^t(C_i) \leftarrow \frac{1}{k}, \forall i = 1, ..., k$
6:     repeat
7:         $t \leftarrow t + 1$
8:         for $i = 1, ..., k$ and $j = 1, ..., n$ do                         ▷ Expectation step
9:             $w_{ij} \leftarrow \frac{f(\boldsymbol{x_j}|\boldsymbol{\mu_i},\boldsymbol{\Sigma_i}).P(C_i)}{\sum_{a=1}^{k} f(\boldsymbol{x_j}|\boldsymbol{\mu_a},\boldsymbol{\Sigma_a}).P(C_a)}$      ▷ posterior probability $P^t(C_i|\boldsymbol{x_j})$
10:        end for
11:        for $i = 1, ..., k$ do                                              ▷ Maximization Step
12:            $\boldsymbol{\mu}_i^t \leftarrow \frac{\sum_{j=1}^{n} w_{ij}\boldsymbol{x_j}}{\sum_{j=1}^{n} w_{ij}}$                                  ▷ re-estimate mean
13:            $\boldsymbol{\Sigma}_i^t \leftarrow \frac{\sum_{j=1}^{n} w_{ij}(\boldsymbol{x_j}-\boldsymbol{\mu_j})(\boldsymbol{x_j}-\boldsymbol{\mu_j})^T}{\sum_{j=1}^{n} w_{ij}}$      ▷ re-estimate covariance matrix
14:            $P^t(C_i) \leftarrow \frac{\sum_{j=1}^{n} w_{ij}}{n}$                                   ▷ re-estimate priors
15:        end for
16:    until $\sum_{i=1}^{k} ||\boldsymbol{\mu}_i^t - \boldsymbol{\mu}_i^{t-1}||^2 \leq \epsilon$
17: end function
```

Problem 1

Implement Expectation-Maximization (EM) algorithm for Gaussian mixture models (see the EM algorithm above) in *R* or *Python* and call this program G_k. As you present your code explain your protocol for [20 points]

1. initializing each Gaussian
2. maintaining k Gaussian
3. deciding ties
4. stopping criteria

R/Python Code

subsectionR/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
from sklearn.preprocessing import StandardScaler
from scipy.stats import multivariate_normal
from sklearn.metrics import silhouette_score, calinski_harabasz_score
```

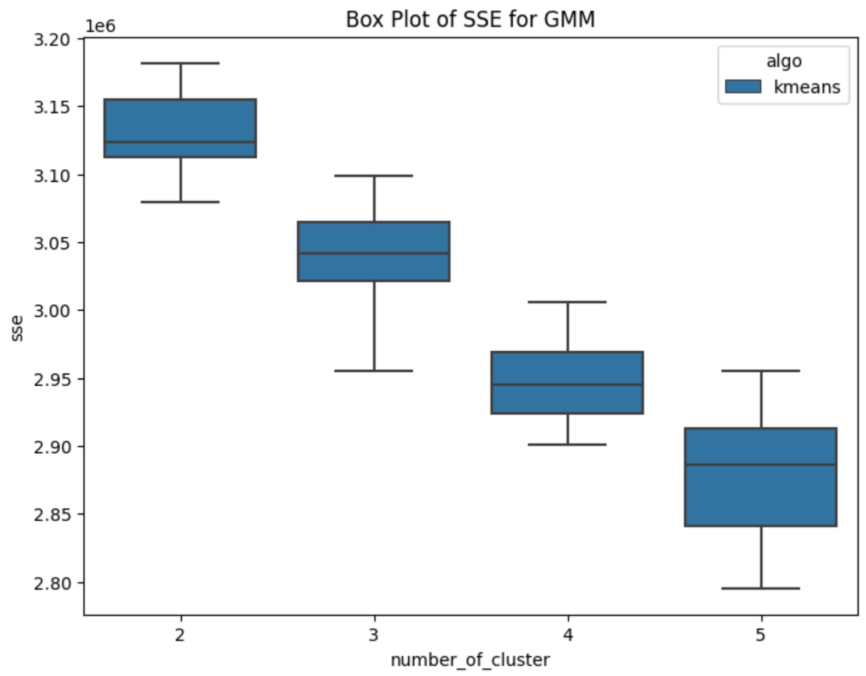
Problem 1 continued on next page. . .

```
from scipy.spatial.distance import euclidean

df= pd.read_csv('dataset_diabetes/diabetic_data.csv')

from sklearn.metrics import silhouette_score,
calinski_harabasz_score
def GMM_initialization(df,k):
    number_of_rows=df.shape[0]
    number_of_columns=df.shape[1]
    means_matrix = df.sample(n=k).values
    identity_matrix=np.eye(number_of_columns)
    covariance_matrix=np.array([identity_matrix]*k)
    weights_matrix=np.array([float(1/k)]*k)
    return
    means_matrix,covariance_matrix,weights_matrix,number_o
    f_rows,number_of_columns


def
calculate_posterior(data,means_matrix,covariance_matrix,we
ights_matrix,k,number_of_columns):
    posterior=np.zeros(k)
    for i in range(k):
        try:
            pseudo_inverse =
            np.linalg.pinv(covariance_matrix[i] +
            np.eye(covariance_matrix[i].shape[0]) * 1e-2,
            rcond=1e-10)
            posterior[i] = multivariate_normal.pdf(data,
            mean=means_matrix[i], cov=pseudo_inverse)
        except Exception as e:
            continue
    return
    posterior*weights_matrix/(posterior*weights_matrix).su
    m()

def maintain_k_clusters(labels,k):
    unique_values, value_counts = np.unique(labels,
    return_counts=True)
    missing_labels=[i for i in range(k) if i not in
    unique_values]
    unique_values_to_be_replaced=[unique_values[i] for i in np.where(value_counts > 1)[0]]

    indices=[i for i in range(len(labels))    if
    labels[i] in unique_values_to_be_replaced]
    random_indices = np.random.choice(indices,
    size=len(missing_labels), replace=False)
    for i,val in enumerate(random_indices):
        labels[val]=missing_labels[i]
    return labels


def sum_of_square_error_em(new_centroids, data, labels):
    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],dtype=np.float64), axis=1)
        #print(distance)
        sse.append(distance.sum())
    # Return the sum of squared errors

    a=sum(sse)
    return a

def Calinski_index_em(df_data,clusters):
    ch_score = calinski_harabasz_score(df_data, clusters)
    return ch_score



def GMM(df_cleaned_dia,k,tao):
    scaler = StandardScaler()
    scaler.fit(df_cleaned_dia)
    scaled_input=scaler.transform(df_cleaned_dia)

    scaled_input_df=
    pd.DataFrame(scaled_input,columns=df_cleaned_dia.colum
    ns)

    means_matrix,covariance_matrix,weights_matrix,number_o
    f_rows,number_of_columns=
    GMM_initialization(scaled_input_df,k)
    likelihood=0
    means_matrix_initial=means_matrix
    for i in range(k):
        try:

            pseudo_inverse =
            np.linalg.pinv(covariance_matrix[i] + np.diag(np.ones(covariance_matrix[i].shape[0])
            * 1e-10))
            likelihood=likelihood+weights_matrix[i]*multiv
            ariate_normal.logpdf(scaled_input,means_matrix
            [i], pseudo_inverse)
        except Exception as e:
            continue
    log_likelihood_old=np.sum(likelihood)
    old_means_matrix_df=pd.DataFrame(means_matrix)
    posterior_probability = np.zeros((scaled_input.shape[0], k))
    iterations=0

    while (True):
        iterations+=1
        # Expectation
        for i in range(scaled_input.shape[0]):
            posterior_probability[i] =
            calculate_posterior(scaled_input[i],
            means_matrix,covariance_matrix,weights_matrix,
            k,number_of_columns)

        # Maximization
        posterior_probability=np.nan_to_num(posterior_prob
        ability, nan=0)
        for i in range(k):
            # Calculating weight
            weight = posterior_probability[:, i].sum()
            #print(weight)
            # Updating each centroid
            means_matrix[i] = (posterior_probability[:,
            i] @ scaled_input) / weight
            #print(1,means_matrix[i])
            # Subtracting the mean value from data
            scaled_input_diff = scaled_input - means_matrix[i]

            # Update the covariance matrix
            covariance_matrix[i] =
            (posterior_probability[:, i] *
            scaled_input_diff.T @ scaled_input_diff) /
            weight

            # Update the weights matrix
            weights_matrix[i] = weight / number_of_rows


        likelihood=0
        for i in range(k):
            try:
                pseudo_inverse =
                np.linalg.pinv(covariance_matrix[i] +
                np.diag(np.ones(covariance_matrix[i].shape
                [0]) * 1e-10))
                likelihood=likelihood+weights_matrix[i]*mu
                ltivariate_normal.logpdf(scaled_input,mean
                s_matrix[i], sudo_inverse)
            except Exception as e:
                continue
        log_likelihood_new =np.sum(likelihood)

        new_means_matrix_df=pd.DataFrame(means_matrix)
        distance = []
        for col in new_means_matrix_df.columns:
            col_distance =
            euclidean(old_means_matrix_df[col],
            new_means_matrix_df[col])
            distance.append(col_distance)
        tao_calculated=sum(distance)/k



        if tao_calculated<
        tao:#log_likelihood_new>log_likelihood_old and
        100*((log_likelihood_new - log_likelihood_old) /
        log_likelihood_old)<tao:

            print("Converged")
            labels=np.argmax(posterior_probability,axis=1)
            labels=maintain_k_clusters(labels,k)
            labels_df=pd.DataFrame(labels)
            means_matrix_df=pd.DataFrame(means_matrix,colu
            mns=scaled_input_df.columns)
            sse=sum_of_square_error_em(means_matrix_df,
            scaled_input_df, labels_df)
            clainski=
            Calinski_index_em(scaled_input_df,labels_df)
            return sse,clainski,means_matrix_initial
        #else:
            #log_likelihood_old=log_likelihood_new

        if iterations>100:
            print("Max iteration reached")
            labels=np.argmax(posterior_probability,axis=1)
            labels=maintain_k_clusters(labels,k)
            labels_df=pd.DataFrame(labels)
            means_matrix_df=pd.DataFrame(means_matrix,colu
            mns=scaled_input_df.columns)
            sse=sum_of_square_error_em(means_matrix_df,
            scaled_input_df, labels_df)
            clainski=
            Calinski_index_em(scaled_input_df,labels_df)
            return sse,clainski,means_matrix_initial

error_matrix_em=[]
error_matrix_kmeans=[]
for i in range(2,6):
    for j in range(1,21):
        sse,clainski,means_matrix_initial=GMM(df_cleaned_dia,i,10)
        error_matrix_em.append([i,sse,clainski])

        sse,clainski=kmeans_lyod_with_error(scaled_input_df,i,10,means_matrix_initial)
        error_matrix_kmeans.append([i,sse,clainski])
error_df_em= pd.DataFrame(error_matrix_em,columns=['number_of_cluster', 'sse','clainski'])
error_df_kmeans= pd.DataFrame(error_matrix_kmeans,columns=['number_of_cluster', 'sse','clainski'])
```

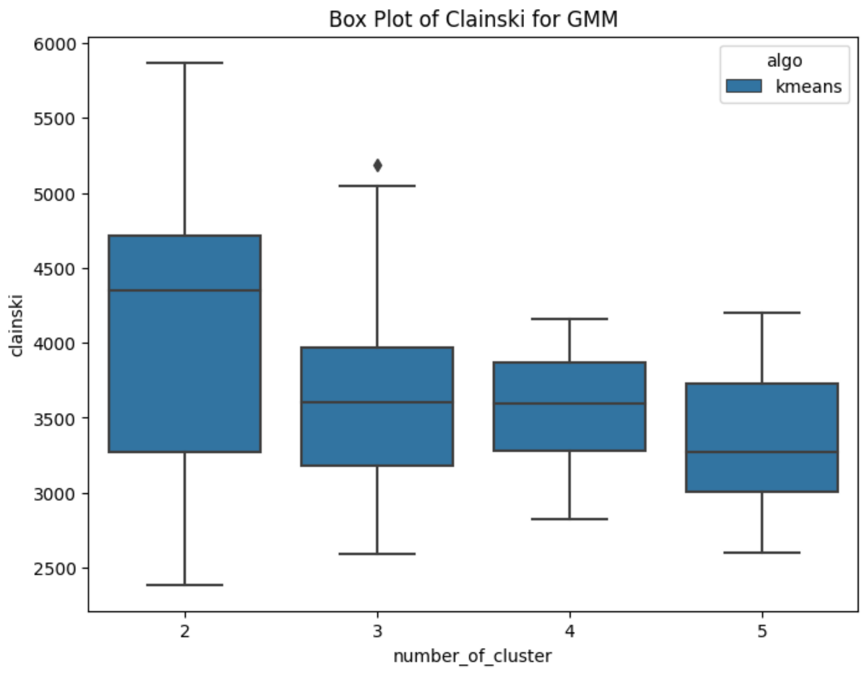
Problem 1 continued on next page. . .

```
    import seaborn as sns

fig, ax = plt.subplots(figsize=(8,6))

sns.boxplot(x='number_of_cluster', y='sse', hue='algo',
            data=run_time_diab[run_time_diab['algo'].isin

            (['kmeans'])],ax=ax);
plt.title('Box Plot of SSE for GMM')
plt.show()

import seaborn as sns

fig, ax = plt.subplots(figsize=(8,6))

sns.boxplot(x='number_of_cluster', y='clainski', hue='algo',
            data=run_time_diab[run_time_diab['algo'].isin (['kmeans'])],ax=ax);
plt.title('Box Plot of Clainski for GMM')
plt.show()
```

Problem 1 continued on next page...



(1)



(2)

The dataset contains information on patients with diabetes who were admitted to 130 hospitals in the United States between 1999 and 2008. The primary purpose of the dataset is to explore features that affect readmission rates for diabetic patients.The dataset can be used for various purposes, including exploring the relationship between various features like patient condition,labtest,medications and readmission rates, developing predictive models for readmission, and evaluating the quality of care provided by hospitals. The dataset can be used to identify the relationship between various parameters and can be used by Doctors,

Problem 1 continued on next page. . .

researchers to identify the mistakes or areas of improvements or what better could have been done , so that patient is not readmitted. Also, identify some pattern or cluster in the data given all the features or identify the main features affecting the readmission. The different data types that the dataset includes are integer and object data. The dataset's features include patient demographics, admission type and source, diagnosis and procedure codes, lab tests, medications, length of stay,etc. The target variable is whether or not the patient was readmitted to the hospital within 30 days of discharge. Overall there are 101766 entries of data and 50 features/columns including the target variable. There are no null entries as such in data. But some of the columns contains ? Replacing question mark with NAN. After doing that we can see the results and now this missing data needs to be handled. race, weight, payercode, medicalspecialty, diag1 , diag2, diag3 Columns contains null data. Dropping the columns where the count of null data is almost around 50 percent as they dont provide significant information.

Discussion of Initialization of Gaussians

Answer here…
GMM (Gaussian Mixture Model) is a paramteric technique. The model is trained using the Expectation-Maximization (EM) algorithm. Before the expectation steps, there are certain assumptions about the data and initializations done, before the start of the actual algorithm. The main assumption is that for all the features, the data is distributed normally and then we use soft clustering to assign each data point to a cluster. The goal of GMM is to estimate the parameters of the underlying mixture model that best fit the observed data. The different vectors initialized are as follows:
1) Mean Vector- The mean vector is initialized by randomly selecting the k data points from the data, considering them as the centroids coordinate. Here k is the number of clusters.
2) Covariance Vector- The covariance vector is initially initialized by creating an identity matrix of size features* features. The vector contains k number of identity matrices where k is the number of clusters.
3) Weight Vector- The weight vector is the weights of each cluster and initially all the clusters have same weight of 1/k where k is number of clusters. So the vector contains k weights each with a value of 1/k.

Discussion of Maintaining k Gaussians

Answer here…
The EM algorithm keeps on switching between an Expectation step and a Maximization step. In the Expectation step, the algorithm estimates the probability that each data point belongs to each component. In the Maximization step, the algorithm updates the parameters of the model to maximize the likelihood of the observed data given the estimated component probabilities. The GMM is a soft clustering method i.e the data point is not rigidly assigned to each cluster, instead, we have a posterior probability vector that stores the likelihood of each point going into a cluster. Hence k clusters are maintained at each iteration. For a data point, the maximum probability for going into a cluster as per the posterior probability calculation is selected. After every iteration, the weights and mean vectors for each cluster are updated.

Discussion of Deciding Ties

Answer here… Here we maintain the posterior probability matrix for each data point and at every iteration. In case the probability of assigning a datapoint to cluster is same, I have used random selection , this means that in case of a tie between selecting a cluster, any conflicting cluster can be selected all having equal probability of getting selected.

Discussion of Stopping Criteria

Answer here…

Problem 1 continued on next page…

As per the algorithm provided, I have implemented a similar logic. When the mean vector is updated and the euclidean difference between the current centroid and the previous centroid is less than a particular threshold, in that case, the algorithm converges, and we get the final centroid coordinates, weight vector, probabilities, etc.

Problem 2

Run your program, G_k, over the Diabetes data set and compare G_k with C_k (your k-means program from homework 4). Click on the below link to download the data set [50 points].

- Diabetes 130-US Hospitals Data Set

Answer the following questions:

1. Initialize G_k and C_k with the same set of initial points (initial centroids for C_k and μ_i-s for G_k are identical) and run them for $k = 2, \ldots, 5$ for 20 runs each. Compare G_k and C_k using two different 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.

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
from sklearn.preprocessing import StandardScaler
from scipy.stats import multivariate_normal
from sklearn.metrics import silhouette_score,
calinski_harabasz_score
from scipy.spatial.distance import euclidean

df= pd.read_csv('dataset_diabetes/diabetic_data.csv')

#Replacing Question Mark with NAN
import numpy as np
df.replace({'?':np.nan},inplace=True)
df.head()

encounter_id   patient_nbr    race gender    age  weight    admission_type_id   discharge_dispos
0    2278392    8222157   Caucasian Female    [0-10)     NaN  6    25   1    1    ...  No   No   N
1    149190     55629189  Caucasian Female    [10-20)    NaN  1    1    7    3    ...  No   Up   N
2    64410      86047875  AfricanAmerican     Female     [20-30)    NaN  1    1    7    2    ...  N
3    500364     82442376  Caucasian Male [30-40)    NaN  1    1    7    2    ...  No   Up   No   N
4    16680      42519267  Caucasian Male [40-50)    NaN  1    1    7    1    ...  No   Steady    N
5 rows    50 columns
```

```

null_feature=[i for i in df.columns if
df[i].isnull().sum()>=1]
print('Null features {} \n'.format(null_feature))
print('Feature \t null_count \t not_null_count')
for i in null_feature:
    print('{} \t {} \t\t
    {}'.format(i,df[i].isnull().sum(),df[i].count()))

Null features ['race', 'weight', 'payer_code', 'medical_specialty', 'diag_1', 'diag_2', 'diag_3'

Feature     null_count      not_null_count
race        2273            99493
weight      98569           3197
payer_code       40256           61510
medical_specialty    49949           51817
diag_1      21         101745
diag_2      358             101408
diag_3      1423            100343


#Dropping columns with count of null values around the
count of not null values. As they dont provide
significant information or
#mostly contains null data
df.drop(['weight','payer_code','medical_specialty'],axis=
1,inplace=True)



df['age'] = df['age'].replace({'[0-10)': 5, '[10-20)':
15, '[20-30)': 25, '[30-40)': 35, '[40-50)': 45, '[50-
60)': 55, '[60-70)': 65, '[70-80)': 75, '[80-90)': 85,
'[90-100)': 95})

df['readmitted'] = df['readmitted'].replace({'>30':1,'<30':1,'NO':0})

for i in df.select_dtypes(include=['int',
'float']).columns.to_list():
    print('The numeric feature is {} \n The value counts
    are {}'.format(i,df[i].value_counts()) )

df.drop(columns=['encounter_id','patient_nbr'],inplace=True,axis=1)

meds=['max_glu_serum', 'A1Cresult', 'metformin',
'repaglinide', 'nateglinide', 'chlorpropamide',
      'glimepiride', 'acetohexamide', 'glipizide',
      'glyburide', 'tolbutamide', 'pioglitazone',
      'rosiglitazone',
      'acarbose', 'miglitol', 'troglitazone',
      'tolazamide', 'examide', 'citoglipton', 'insulin',
      'glyburide-metformin',
      'glipizide-metformin', 'glimepiride-pioglitazone',
     'metformin-rosiglitazone', 'metformin-pioglitazone']
df_value_counts=pd.DataFrame()
for i in meds:
    value_counts=df[i].value_counts()
    percent = []
    for j in value_counts.index:
        percent.append(value_counts[j] *100/ len(df))
    ## PErcentage dataframe to store the feature, its
    unique values, the count and the percentage
    df_temp=pd.DataFrame({'Feature':i,'Value':
    value_counts.index, 'Count': value_counts.values,
    'Percentage': percent})
    df_value_counts=pd.concat([df_value_counts,df_temp],I
    gnore_index=True)
df_value_counts.head(80)

Feature   Value     Count     Percentage
0    max_glu_serum  None 96420     94.746772
1    max_glu_serum  Norm 2597 2.551933
2    max_glu_serum  >200 1485 1.459230
3    max_glu_serum  >300 1264 1.242065
4    A1Cresult None 84748     83.277322
...  ...  ...  ...  ...
74   glimepiride-pioglitazone Steady    1    0.000983
75   metformin-rosiglitazone  No   101764    99.998035
76   metformin-rosiglitazone  Steady    2    0.001965
77   metformin-pioglitazone   No   101765    99.999017
78   metformin-pioglitazone   Steady    1    0.000983
79 rows    4 columns

skewed_data=df_value_counts[df_value_counts['Percentage']
>95]['Feature'].to_list()
df.drop(columns=skewed_data,inplace=True)

label_encoded_columns=[]
for i in df.select_dtypes(include=
['object']).columns.to_list():
    if i not in skewed_data:
        label_encoded_columns.append(i)
## The columns remaining after all the EDA that are to be label encoded


df_cleaned_dia=df.copy()
## creating the copy of data before performing label encoding and one hot encoding
one_hot = pd.get_dummies(df_cleaned_dia[['gender','race']])
label_encoded_columns.remove('diag_1')
label_encoded_columns.remove('diag_2')
label_encoded_columns.remove('diag_3')
label_encoded_columns.remove('gender')
label_encoded_columns.remove('race')
# combine the one-hot encoded columns with the original dataframe
df_cleaned_dia = pd.concat([df_cleaned_dia, one_hot], axis=1)
df_cleaned_dia.drop(columns=['diag_1','diag_2','diag_3','gender','race'],inplace=True)
```

```
##
df_cleaned_dia[label_encoded_columns]=df_cleaned_dia[labe
l_encoded_columns].swifter.apply(LabelEncoder().fit_trans
form)




from sklearn.metrics import silhouette_score,
calinski_harabasz_score
def GMM_initialization(df,k):
    number_of_rows=df.shape[0]
    number_of_columns=df.shape[1]
    means_matrix = df.sample(n=k).values
    identity_matrix=np.eye(number_of_columns)
    covariance_matrix=np.array([identity_matrix]*k)
    weights_matrix=np.array([float(1/k)]*k)
    return
    means_matrix,covariance_matrix,weights_matrix,number_
    of_rows,number_of_columns


def
calculate_posterior(data,means_matrix,covariance_matrix,w
eights_matrix,k,number_of_columns):
    posterior=np.zeros(k)
    for i in range(k):
        try:
            pseudo_inverse =
            np.linalg.pinv(covariance_matrix[i] +
            np.eye(covariance_matrix[i].shape[0]) * 1e-
            2,
            rcond=1e-10)
            posterior[i] = multivariate_normal.pdf(data,
            mean=means_matrix[i], cov=pseudo_inverse)
        except Exception as e:
            continue
    return posterior*weights_matrix/(posterior*weights_matrix).sum()

def maintain_k_clusters(labels,k):
    unique_values, value_counts = np.unique(labels,
    return_counts=True)
    missing_labels=[i for i in range(k) if i not in
    unique_values]
    unique_values_to_be_replaced=[unique_values[i] for i in np.where(value_counts > 1)[0]]

    indices=[i for i in range(len(labels))    if
    labels[i] in unique_values_to_be_replaced]
    random_indices = np.random.choice(indices,
    size=len(missing_labels), replace=False)
    for i,val in enumerate(random_indices):
        labels[val]=missing_labels[i]
    return labels
```

```

def sum_of_square_error_em(new_centroids, data, labels):
    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],dtype=np.float64), axis=1)
        #print(distance)
        sse.append(distance.sum())
    # Return the sum of squared errors

    a=sum(sse)
    return a

def Calinski_index_em(df_data,clusters):
    ch_score = calinski_harabasz_score(df_data, clusters)
    return ch_score



def GMM(df_cleaned_dia,k,tao):
    scaler = StandardScaler()
    scaler.fit(df_cleaned_dia)
    scaled_input=scaler.transform(df_cleaned_dia)

    scaled_input_df= pd.DataFrame(scaled_input,columns=df_cleaned_dia.colu
    mns)

    means_matrix,covariance_matrix,weights_matrix,number_of_rows,number_of_columns=
    GMM_initialization(scaled_input_df,k)
    likelihood=0
    means_matrix_initial=means_matrix
    for i in range(k):
        try:

            pseudo_inverse =
            np.linalg.pinv(covariance_matrix[i] + np.diag(np.ones(covariance_matrix[i].shape[0]
            )* 1e-10))
            likelihood=likelihood+weights_matrix[i]*multi
        variate_normal.logpdf(scaled_input,means_matrix
            [i], pseudo_inverse)
        except Exception as e:
            continue
    log_likelihood_old=np.sum(likelihood)
    old_means_matrix_df=pd.DataFrame(means_matrix)
    posterior_probability = np.zeros((scaled_input.shape[0], k))
    iterations=0

    while (True):
        iterations+=1
        # Expectation
        for i in range(scaled_input.shape[0]):
            posterior_probability[i] =
            calculate_posterior(scaled_input[i],
            means_matrix,covariance_matrix,weights_matrix,
            k,number_of_columns)

        # Maximization
        posterior_probability=np.nan_to_num(posterior_prob
        ability, nan=0)
        for i in range(k):
            # Calculating weight
            weight = posterior_probability[:, i].sum()
            #print(weight)
            # Updating each centroid
            means_matrix[i] = (posterior_probability[:,
            i] @ scaled_input) / weight
            #print(1,means_matrix[i])
            # Subtracting the mean value from data
            scaled_input_diff = scaled_input - means_matrix[i]

            # Update the covariance matrix
            covariance_matrix[i] =
            (posterior_probability[:, i] *
            scaled_input_diff.T @ scaled_input_diff) /
            weight

            # Update the weights matrix
            weights_matrix[i] = weight / number_of_rows


        likelihood=0
        for i in range(k):
            try:
                pseudo_inverse =
                np.linalg.pinv(covariance_matrix[i] +
                np.diag(np.ones(covariance_matrix[i].shape[0]) * 1e-10))
                likelihood=likelihood+weights_matrix[i]*m
                ultivariate_normal.logpdf(scaled_input,me
                ans_matrix[i], sudo_inverse)
            except Exception as e:
                continue
        log_likelihood_new =np.sum(likelihood)

        new_means_matrix_df=pd.DataFrame(means_matrix)
        distance = []
```

Problem 2 continued on next page. . .

```
        for col in new_means_matrix_df.columns:
            col_distance =
            euclidean(old_means_matrix_df[col],
            new_means_matrix_df[col])
            distance.append(col_distance)
        tao_calculated=sum(distance)/k



        if tao_calculated<
        tao:#log_likelihood_new>log_likelihood_old and
        100*((log_likelihood_new - log_likelihood_old) /
        log_likelihood_old)<tao:

            print("Converged")
            labels=np.argmax(posterior_probability,axis=1
            )
            labels=maintain_k_clusters(labels,k)
            labels_df=pd.DataFrame(labels)
            means_matrix_df=pd.DataFrame(means_matrix,col
            umns=scaled_input_df.columns)
            sse=sum_of_square_error_em(means_matrix_df,
            scaled_input_df, labels_df)
            clainski=
            Calinski_index_em(scaled_input_df,labels_df)
            return sse,clainski,means_matrix_initial
        #else:
            #log_likelihood_old=log_likelihood_new

        if iterations>100:
            print("Max iteration reached")
            labels=np.argmax(posterior_probability,axis=1
            )
            labels=maintain_k_clusters(labels,k)
            labels_df=pd.DataFrame(labels)
            means_matrix_df=pd.DataFrame(means_matrix,col
            umns=scaled_input_df.columns)
            sse=sum_of_square_error_em(means_matrix_df,
            scaled_input_df, labels_df)
            clainski=
            Calinski_index_em(scaled_input_df,labels_df)
            return sse,clainski,means_matrix_initial


import time
from scipy.spatial.distance import euclidean
def initialize_centroids(df, k,means_matrix):
    """
    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=pd.DataFrame(means_matrix,columns=df.column
    s)
    centroids=centroids.T
    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
```

Problem 2 continued on next page. . .

```
def error_clusters(df_new_centroids,df1,df_label):
    """
    Calculate the error rate of each cluster.

    Args:
    - df_label (pandas.DataFrame): the label of the
    nearest centroid for each data point.
    - df1 (pandas.DataFrame): the dataset.
    - df_new_centroids (pandas.DataFrame): The new centroids computed in the current iteration.

    Returns:
    - error_rate (float): the total error rate of all clusters.
    """


    #Calculate mean value
    mean_centroid=df1.groupby('readmitted').mean().reset_index()
    # Transpose the new centroids dataframe and reset the index
    new_centroids= df_new_centroids.T
    # Get the columns of the data dataframe
    columns = df1.columns

    sse = []
    # Compute the distance between each data point and its assigned centroid
    for i in range(len(new_centroids)):    #### centroid
        s=[]
        for j in range(len(mean_centroid)): ### mean centroid
        # Compute the distance between each data point and its assigned centroid
            distance =
            np.sum(np.square(mean_centroid[mean_centroid[
            'readmitted']==j][columns] -
            new_centroids.iloc[i][columns]), axis=1)
            s.append(distance.iloc[0])
        sse.append(s)
    ## key  is the cluster number and value is the merged value
    merge_label=pd.DataFrame(sse).idxmin(axis=1).to_dict()
    ## Merging cluster based on the target variable
    df_label[0]=df_label[0].replace(merge_label)

    df1 = df1.join(df_label) # add the label column to the dataset
    df1.rename(columns={0: 'Label'}, inplace=True) # rename the label column
    error_list = []
    for i in df1['Label'].value_counts().index:
        df_cluster = df1[df1['Label'] == i] # filter the
        dataset to include only the data points in the
        current cluster
        y = len(df_cluster[df_cluster['readmitted'] ==
        1]) # count the number of data points in the
        current cluster that were readmitted
        n = len(df_cluster[df_cluster['readmitted'] ==
        0]) # count the number of data points in the
        current cluster that were not readmitted
        if y == 0 and n == 0:
```

Problem 2 continued on next page. . .

```
            error = 0
        else:
            error = n / (n + y) # calculate the error
            rate of the current cluster
        error_list.append(error)
    return round(sum(error_list),4)


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 = new_centroids.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 Calinski_index(df_data,clusters):
    ch_score = calinski_harabasz_score(df_data, clusters)
    return ch_score




def kmeans_lyod_with_error(df1, k, tou,means_matrix_initial):
    """
    Function to run the K-means Lloyd algorithm.
    Input:
        - df1: pandas dataframe with the data
        - k: integer number of clusters
```

Problem 2 continued on next page. . .

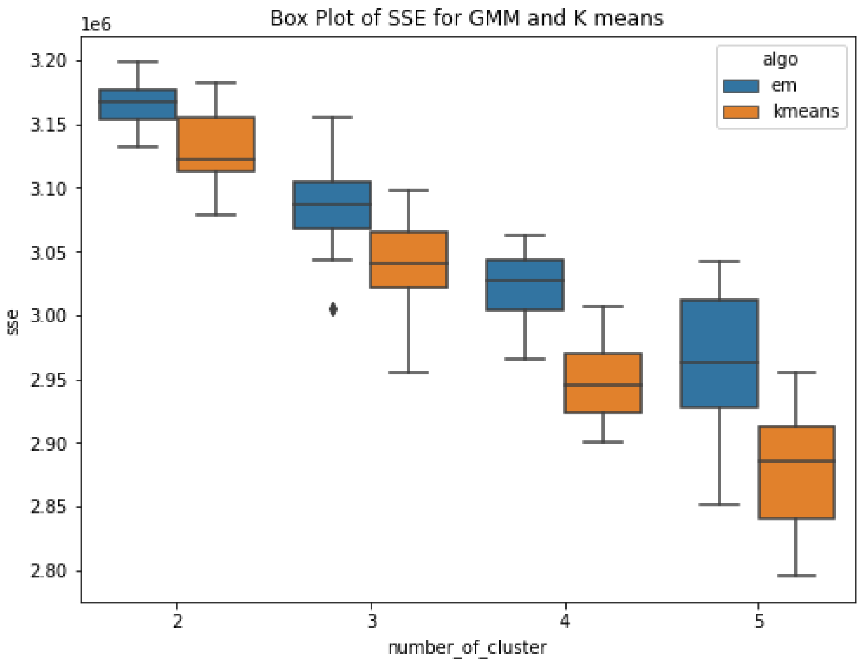
```
        - tou: float tolerance level to stop the algorithm
    Output:
        - centroids: pandas dataframe with the final centroids as columns and index as label
    """
    start_time=time.time()
    centroids = initialize_centroids(df1,
    k,means_matrix_initial) # Initialize random centroids
    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()
        clainski= Calinski_index(df1,df_label)
        if iteration>100:
            print("Iteration exceeded")

            return sse,clainski
            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,clainski
            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



scaler = StandardScaler()
```

```
scaler.fit(df_cleaned_dia)
scaled_input=scaler.transform(df_cleaned_dia)

scaled_input_df= pd.DataFrame(scaled_input,columns=df_cleaned_dia.columns)



error_matrix_em=[]
error_matrix_kmeans=[]
for i in range(2,6):
    for j in range(1,21):
        sse,clainski,means_matrix_initial=GMM(df_cleaned_
        dia,i,10)
        error_matrix_em.append([i,sse,clainski])

        sse,clainski=kmeans_lyod_with_error(scaled_input_
        df,i,10,means_matrix_initial)
        error_matrix_kmeans.append([i,sse,clainski])
error_df_em= pd.DataFrame(error_matrix_em,columns=['number_of_cluster', 'sse','clainski'])
error_df_kmeans= pd.DataFrame(error_matrix_kmeans,columns=
['number_of_cluster', 'sse','clainski'])




error_df_em.to_csv('6_em.csv',index=False)
error_df_kmeans.to_csv('6_kmeans.csv',index=False)




error_df_em['algo']='em'
error_df_kmeans['algo']='kmeans'



run_time_diab=pd.DataFrame()
run_time_diab=pd.concat( [
error_df_em[['algo','number_of_cluster','sse','clainski']],
    error_df_kmeans[['algo','number_of_cluster',
    'sse','clainski']]
                        ],ignore_index=True )

import seaborn as sns

fig, ax = plt.subplots(figsize=(8,6))

sns.boxplot(x='number_of_cluster', y='sse', hue='algo',

            data=run_time_diab[run_time_diab['algo'].isin
            (['kmeans','em'])],ax=ax);
plt.title('Box Plot of SSE for GMM and K means')
plt.show()


import seaborn as sns
```

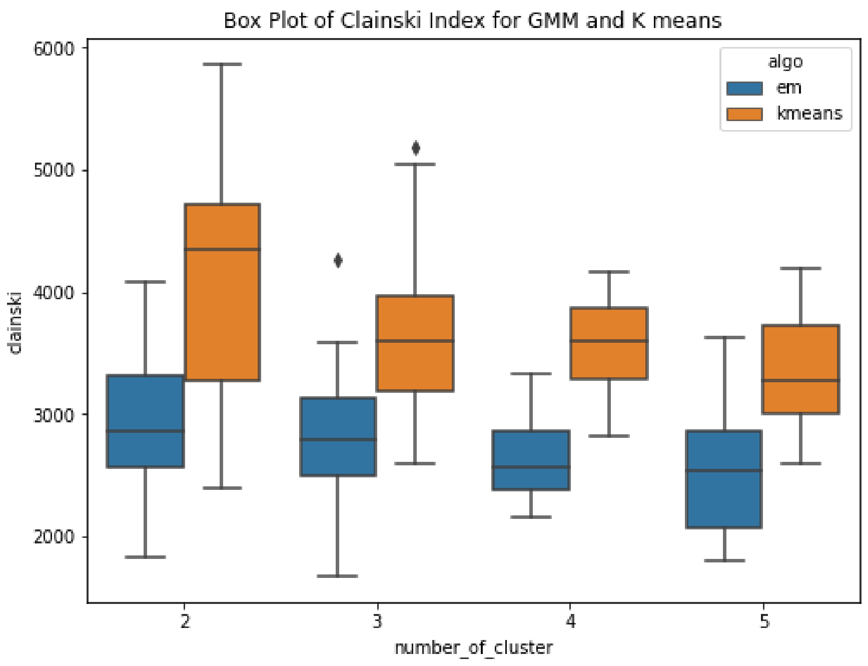
```
fig, ax = plt.subplots(figsize=(8,6))

sns.boxplot(x='number_of_cluster', y='clainski', hue='algo',
            data=run_time_diab[run_time_diab['algo'].isin (['kmeans','em'])],ax=ax);
plt.title('Box Plot of Clainski Index for GMM and K means')
plt.show()
```

Plot/s

Place images here with suitable captions.



(3)



(4)

Discussion of Experiments

Answer here...
The GMM (EM) algorithm and K-means algorithm is ran over the diabetes dataset 20 times each for clusters ranging from 2 to 5.Have used within sum of square error and Calinski-Harabasz score to evaluate the clustering techniques. Within Sum of Squares (WSS), which calculates the total sum of distances between each point in a cluster and its centroid. Calinski-Harabasz score (CHS), which evaluates the quality of the clustering solution by comparing the separation between clusters to the dispersion within clusters. From the box plots, it can be seen that the median within the sum of square error for k means is less than the median within sse of EM. This means for the given dataset, K means have performed better or the clusters are well separated and the within-cluster the data points are very close to the centroid. Whereas for GMM, it has higher sse, one of the reasons could be the clusters formed are overlapping and this may lead to higher error. This can be confirmed by the Calinski Harabasz score (CHS). The median CHS score for kmeans is higher than that of the GMM for all the cluster, showing the clusters formed by Kmeans are well seperated. We can use other techniques to perform the validity and this highly depends upon the requirements and use case.

Problem 2 continued on next page...

2. Run your G_k without updating the covariance matrices and priors across iterations. Compare G_k and C_k using two different 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.

R or Python script

```
# Sample R 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
from sklearn.preprocessing import StandardScaler
from scipy.stats import multivariate_normal
from sklearn.metrics import silhouette_score, calinski_harabasz_score
from scipy.spatial.distance import euclidean


def GMM_initialization(df,k):
    number_of_rows=df.shape[0]
    number_of_columns=df.shape[1]
    means_matrix = df.sample(n=k).values
    identity_matrix=np.eye(number_of_columns)
    covariance_matrix=np.array([identity_matrix]*k)
    weights_matrix=np.array([float(1/k)]*k)
    return means_matrix,covariance_matrix,weights_matrix,number_
    of_rows,number_of_columns


def calculate_posterior(data,means_matrix,covariance_matrix,w
eights_matrix,k,number_of_columns):
    posterior=np.zeros(k)
    for i in range(k):
        try:
            pseudo_inverse =
            np.linalg.pinv(covariance_matrix[i] +
            np.eye(covariance_matrix[i].shape[0]) * 1e-
            2, rcond=1e-10)
            posterior[i] = multivariate_normal.pdf(data,
            mean=means_matrix[i], cov=pseudo_inverse)
        except Exception as e:
            continue
    return
    posterior*weights_matrix#/(posterior*weights_matrix).
    sum()

def maintain_k_clusters(labels,k):
    unique_values, value_counts = np.unique(labels,
```

Problem 2 continued on next page. . .

```
    return_counts=True)
    missing_labels=[i for i in range(k) if i not in
    unique_values]
    unique_values_to_be_replaced=[unique_values[i] for i
    in np.where(value_counts > 1)[0]]

    indices=[i for i in range(len(labels))    if
    labels[i] in unique_values_to_be_replaced]
    random_indices = np.random.choice(indices,
    size=len(missing_labels), replace=False)
    for i,val in enumerate(random_indices):
        labels[val]=missing_labels[i]
    return labels


def sum_of_square_error_em(new_centroids, data, labels):
    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],dtype=np.float64), axis=1)
        #print(distance)
        sse.append(distance.sum())
    # Return the sum of squared errors

    a=sum(sse)
    return a

def Calinski_index_em(df_data,clusters):
    ch_score = calinski_harabasz_score(df_data, clusters)
    return ch_score



def GMM(df_cleaned_dia,k,tao):
    scaler = StandardScaler()
    scaler.fit(df_cleaned_dia)
    scaled_input=scaler.transform(df_cleaned_dia)

    scaled_input_df= pd.DataFrame(scaled_input,columns=df_cleaned_dia.colu
    mns)

    means_matrix,covariance_matrix,weights_matrix,number_of_rows,number_of_columns=
    GMM_initialization(scaled_input_df,k)
```

Problem 2 continued on next page. . .

```
    likelihood=0
    means_matrix_initial=means_matrix
    for i in range(k):
        try:

            pseudo_inverse =
            np.linalg.pinv(covariance_matrix[i] +
            np.diag(np.ones(covariance_matrix[i].shape[0]
            ) * 1e-10))
            likelihood=likelihood+weights_matrix[i]*multi
            variate_normal.logpdf(scaled_input,means_matr
            ix[i], pseudo_inverse)
        except Exception as e:
            continue
    log_likelihood_old=np.sum(likelihood)
    old_means_matrix_df=pd.DataFrame(means_matrix)
    posterior_probability =
    np.zeros((scaled_input.shape[0], k))
    iterations=0

    while (True):
        iterations+=1
        # Expectation
        for i in range(scaled_input.shape[0]):
            posterior_probability[i] =
            calculate_posterior(scaled_input[i],
            means_matrix,covariance_matrix,weights_matrix
            ,k,number_of_columns)

        # Maximization
        posterior_probability=np.nan_to_num(posterior_pro
        bability, nan=0)
        for i in range(k):
            # Calculating weight
            weight = posterior_probability[:, i].sum()
            #print(weight)
            # Updating each centroid
            means_matrix[i] = (posterior_probability[:, i] @ scaled_input) / weight
            #print(1,means_matrix[i])
            # Subtracting the mean value from data
            scaled_input_diff = scaled_input - means_matrix[i]

            # Update the covariance matrix
            #covariance_matrix[i] = (posterior_probability[:, i] *
            scaled_input_diff.T @ scaled_input_diff) / weight

            # Update the weights matrix
            #weights_matrix[i] = weight / number_of_rows


        likelihood=0
        for i in range(k):
            try:
```

Problem 2 continued on next page. . .

```
                pseudo_inverse =
                np.linalg.pinv(covariance_matrix[i] +
                np.diag(np.ones(covariance_matrix[i].shap
                e[0]) * 1e-10))
                likelihood=likelihood+weights_matrix[i]*m
                ultivariate_normal.logpdf(scaled_input,me
                ans_matrix[i], sudo_inverse)
            except Exception as e:
                continue
        log_likelihood_new =np.sum(likelihood)

        new_means_matrix_df=pd.DataFrame(means_matrix)
        distance = []
        for col in new_means_matrix_df.columns:
            col_distance =
            euclidean(old_means_matrix_df[col],
            new_means_matrix_df[col])

            distance.append(col_distance)
        tao_calculated=sum(distance)/k



        if tao_calculated<
        tao:#log_likelihood_new>log_likelihood_old and
        100*((log_likelihood_new - log_likelihood_old) /
        log_likelihood_old)<tao:

            print("Converged")
            labels=np.argmax(posterior_probability,axis=1
            )
            labels=maintain_k_clusters(labels,k)
            labels_df=pd.DataFrame(labels)
            means_matrix_df=pd.DataFrame(means_matrix,col
            umns=scaled_input_df.columns)
            sse=sum_of_square_error_em(means_matrix_df,
            scaled_input_df, labels_df)
            clainski=
            Calinski_index_em(scaled_input_df,labels_df)
            return sse,clainski,means_matrix_initial
        #else:
            #log_likelihood_old=log_likelihood_new

        if iterations>100:
            print("Max iteration reached")
            labels=np.argmax(posterior_probability,axis=1
            )
            labels=maintain_k_clusters(labels,k)
            labels_df=pd.DataFrame(labels)
            means_matrix_df=pd.DataFrame(means_matrix,col
            umns=scaled_input_df.columns)
            sse=sum_of_square_error_em(means_matrix_df,
            scaled_input_df, labels_df)
```

Problem 2 continued on next page. . .

```
            clainski=
            Calinski_index_em(scaled_input_df,labels_df)
            return sse,clainski,means_matrix_initial



import time
from scipy.spatial.distance import euclidean
def initialize_centroids(df, k,means_matrix):
    """
    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=pd.DataFrame(means_matrix,columns=df.column
    s)
    centroids=centroids.T
    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 error_clusters(df_new_centroids,df1,df_label):
    """
    Calculate the error rate of each cluster.

    Args:
    - df_label (pandas.DataFrame): the label of the
    nearest centroid for each data point.
    - df1 (pandas.DataFrame): the dataset.
    - df_new_centroids (pandas.DataFrame): The new centroids computed in the current iteration.

    Returns:
    - error_rate (float): the total error rate of all clusters.
    """


    #Calculate mean value
    mean_centroid=df1.groupby('readmitted').mean().reset_index()
    # Transpose the new centroids dataframe and reset the index
    new_centroids= df_new_centroids.T
    # Get the columns of the data dataframe
    columns = df1.columns

    sse = []
    # Compute the distance between each data point and its assigned centroid
    for i in range(len(new_centroids)):   #### centroid
        s=[]
        for j in range(len(mean_centroid)): ### mean centroid
        # Compute the distance between each data point and its assigned centroid
            distance =
            np.sum(np.square(mean_centroid[mean_centroid[
            'readmitted']==j][columns] -
            new_centroids.iloc[i][columns]), axis=1)
            s.append(distance.iloc[0])
        sse.append(s)
    ## key  is the cluster number and value is the merged value
    merge_label=pd.DataFrame(sse).idxmin(axis=1).to_dict()
    ## Merging cluster based on the target variable
    df_label[0]=df_label[0].replace(merge_label)
```

Problem 2 continued on next page. . .

```
    df1 = df1.join(df_label) # add the label column to the dataset
    df1.rename(columns={0: 'Label'}, inplace=True) # rename the label column
    error_list = []
    for i in df1['Label'].value_counts().index:
        df_cluster = df1[df1['Label'] == i] # filter the
        dataset to include only the data points in the
        current cluster
        y = len(df_cluster[df_cluster['readmitted'] ==
        1]) # count the number of data points in the
        current cluster that were readmitted
        n = len(df_cluster[df_cluster['readmitted'] ==
        0]) # count the number of data points in the
        current cluster that were not readmitted
        if y == 0 and n == 0:
            error = 0
        else:
            error = n / (n + y) # calculate the error
            rate of the current cluster
        error_list.append(error)
    return round(sum(error_list),4)


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 = new_centroids.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
```

Problem 2 continued on next page. . .

```
    return sum(sse)

def Calinski_index(df_data,clusters):
    ch_score = calinski_harabasz_score(df_data, clusters)
    return ch_score




def kmeans_lyod_with_error(df1, k, tou,means_matrix_initial):
    """
    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,means_matrix_initial) # 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()
        clainski= Calinski_index(df1,df_label)
        if iteration>100:
            print("Iteration exceeded")

            return sse,clainski
            break
```

Problem 2 continued on next page. . .

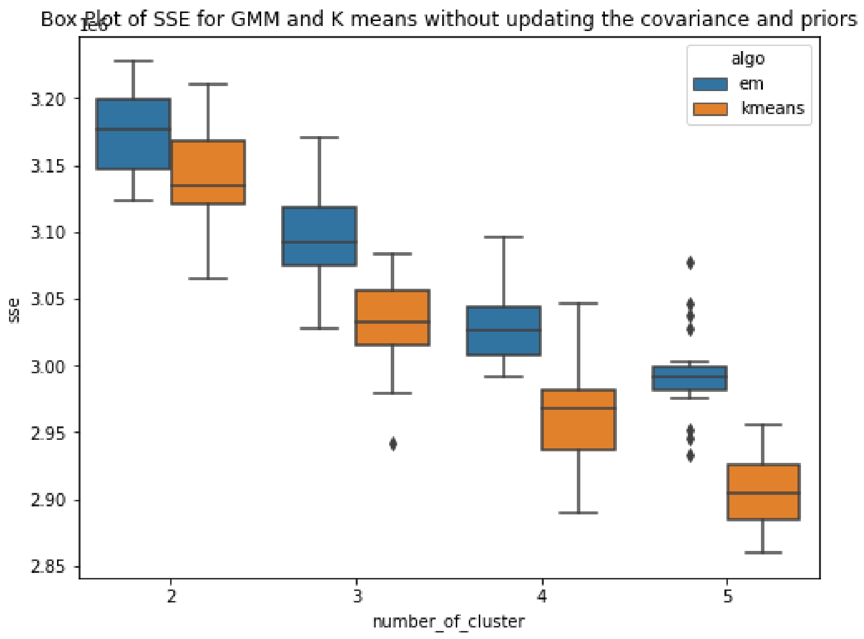
```
        if tao_calculated<tou or iteration >100:   #if
        the convergence is met, kmeans will stop  or
        else if the convergence is never met, after 100
        iteration code will stop
            return  sse,clainski
            break                                   # otherwise indefinite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



scaler = StandardScaler()
scaler.fit(df_cleaned_dia)
scaled_input=scaler.transform(df_cleaned_dia)

scaled_input_df= pd.DataFrame(scaled_input,columns=df_cleaned_dia.columns)


error_matrix_em=[]
error_matrix_kmeans=[]
for i in range(2,6):
    for j in range(1,21):
        sse,clainski,means_matrix_initial=GMM(df_cleaned_
        dia,i,10)
        error_matrix_em.append([i,sse,clainski])

        sse,clainski=kmeans_lyod_with_error(scaled_input_
        df,i,10,means_matrix_initial)
        error_matrix_kmeans.append([i,sse,clainski])
error_df_em= pd.DataFrame(error_matrix_em,columns=['number_of_cluster', 'sse','clainski'])
error_df_kmeans= pd.DataFrame(error_matrix_kmeans,columns=
['number_of_cluster', 'sse','clainski'])


error_df_em.to_csv('6_em_no_update.csv',index=False)
error_df_em.to_csv('6_kmeans_no_update.csv',index=False)


error_df_em['algo']='em'
error_df_kmeans['algo']='kmeans'



run_time_diab=pd.DataFrame()
run_time_diab=pd.concat( [ error_df_em[['algo','number_of_cluster','sse','clainski']],
    error_df_kmeans[['algo','number_of_cluster', 'sse','clainski']]
                         ],ignore_index=True )

import seaborn as sns

fig, ax = plt.subplots(figsize=(8,6))
```

Problem 2 continued on next page. . .

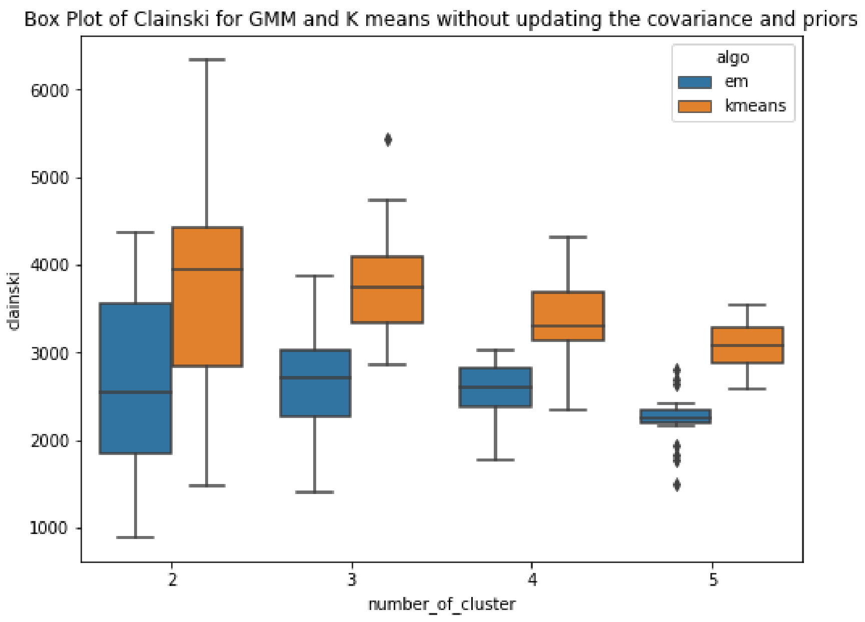
```
sns.boxplot(x='number_of_cluster', y='sse', hue='algo',
            data=run_time_diab[run_time_diab['algo'].isin (['kmeans','em'])],ax=ax);
plt.title('Box Plot of SSE for GMM and K means without updating the covariance and priors')
plt.show()

import seaborn as sns

fig, ax = plt.subplots(figsize=(8,6))

sns.boxplot(x='number_of_cluster', y='clainski', hue='algo',
            data=run_time_diab[run_time_diab['algo'].isin (['kmeans','em'])],ax=ax);
plt.title('Box Plot of Clainski for GMM and K means without updating the covariance and priors')
plt.show()
```

Plot/s

Place images here with suitable captions.



(5)



(6)

Discussion of Experiments

Answer here...
If the GMM algorithm doesn't update the covariances and weights vector, K-means and GMM can converge at the same time. This is because when the covariance matrix is kept fixed, the GMM algorithm becomes similar to K-means, which assigns data points to the nearest centroid based on the Euclidean distance. When the covariances are fixed in GMM, the algorithm still estimates the mean values and mixing coefficients (weights vector) using the EM algorithm. In the E-step, the algorithm calculates the probability of each data point belonging to each Gaussian distribution based on their mean values and fixed covariance matrix. The M-step then updates the mean values and mixing coefficients based on the calculated probabilities. However, this process resembles the K-means algorithm, and both algorithms may converge at nearly similar points if the covariances and weights vector are not updated during the GMM algorithm. This can be seen from the box plot of within sum of square error. The median within SSE of kmeans is slightly less than that of the GMM, but it is comparable. The errors are almost similar, showing convergence. Also the box plot of CHS score shows kmeans and GMM have almost similar values with kmeans more on the positive side of the graph, conveying the same story of convergence when the weights and covariance vectors are not updated.

3. Perform PCA over the Diabetes data set. Create a new data set, Δ_R, with using 90% of the variance. Compare G_k and C_k over Δ_R using two different 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.

R or Python script

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

Problem 2 continued on next page...

```
# 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 values in
    descending order
    sorted_index=eigen_values.argsort()[::-1]
    df_variance=pd.DataFrame(eigen_values[sorted_index]/s
    um(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_vec
    tors) #Performing dot product
    principal_component.columns=[f'PC{i+1}' for i in
```

Problem 2 continued on next page. . .

```
    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_comp
    onent['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['cumul
    ative_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
    _var,eigen_values, eigen_vectors

principal_component,loadings,df_variance,df_number_pc_var
,eigen_values, eigen_vectors=PCA(df_cleaned_dia,0.9)

df_cleaned_dia=
pd.DataFrame(np.real(principal_component.values),
columns=principal_component.columns)
```

```
from sklearn.metrics import silhouette_score,
calinski_harabasz_score
def GMM_initialization(df,k):
    number_of_rows=df.shape[0]
    number_of_columns=df.shape[1]
    means_matrix = df.sample(n=k).values
    identity_matrix=np.eye(number_of_columns)
    covariance_matrix=np.array([identity_matrix]*k)
    weights_matrix=np.array([float(1/k)]*k)
    return
    means_matrix,covariance_matrix,weights_matrix,number_
    of_rows,number_of_columns


def
calculate_posterior(data,means_matrix,covariance_matrix,w
eights_matrix,k,number_of_columns):
    posterior=np.zeros(k)
    for i in range(k):
        try:
            pseudo_inverse =
            np.linalg.pinv(covariance_matrix[i] +
            np.eye(covariance_matrix[i].shape[0]) * 1e-
            2,
            rcond=1e-10)
            posterior[i] = multivariate_normal.pdf(data,
            mean=means_matrix[i], cov=pseudo_inverse)
        except Exception as e:
            continue
    return posterior*weights_matrix/(posterior*weights_matrix).sum()

def maintain_k_clusters(labels,k):
    unique_values, value_counts = np.unique(labels,
    return_counts=True)
    missing_labels=[i for i in range(k) if i not in
    unique_values]
    unique_values_to_be_replaced=[unique_values[i] for i in np.where(value_counts > 1)[0]]

    indices=[i for i in range(len(labels))    if
    labels[i] in unique_values_to_be_replaced]
    random_indices = np.random.choice(indices,
    size=len(missing_labels), replace=False)
    for i,val in enumerate(random_indices):
        labels[val]=missing_labels[i]
    return labels


def sum_of_square_error_em(new_centroids, data, labels):
    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'
```

Problem 2 continued on next page...

```
    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],dtype=np.float64), axis=1)
        #print(distance)
        sse.append(distance.sum())
    # Return the sum of squared errors

    a=sum(sse)
    return a

def Calinski_index_em(df_data,clusters):
    ch_score = calinski_harabasz_score(df_data, clusters)
    return ch_score



def GMM(df_cleaned_dia,k,tao):
    scaler = StandardScaler()
    scaler.fit(df_cleaned_dia)
    scaled_input=scaler.transform(df_cleaned_dia)

    scaled_input_df= pd.DataFrame(scaled_input,columns=df_cleaned_dia.colu
    mns)

    means_matrix,covariance_matrix,weights_matrix,number_of_rows,number_of_columns=
    GMM_initialization(scaled_input_df,k)
    likelihood=0
    means_matrix_initial=means_matrix
    for i in range(k):
        try:

            pseudo_inverse =
            np.linalg.pinv(covariance_matrix[i] + np.diag(np.ones(covariance_matrix[i].shape[0]
            )* 1e-10))
            likelihood=likelihood+weights_matrix[i]*multi
        variate_normal.logpdf(scaled_input,means_matrix
            [i], pseudo_inverse)
        except Exception as e:
            continue
    log_likelihood_old=np.sum(likelihood)
    old_means_matrix_df=pd.DataFrame(means_matrix)
    posterior_probability = np.zeros((scaled_input.shape[0], k))
    iterations=0

    while (True):
        iterations+=1
        # Expectation
        for i in range(scaled_input.shape[0]):
            posterior_probability[i] =
            calculate_posterior(scaled_input[i],
            means_matrix,covariance_matrix,weights_matrix,
            k,number_of_columns)

        # Maximization
        posterior_probability=np.nan_to_num(posterior_prob
        ability, nan=0)
        for i in range(k):
            # Calculating weight
            weight = posterior_probability[:, i].sum()
            #print(weight)
            # Updating each centroid
            means_matrix[i] = (posterior_probability[:,
            i] @ scaled_input) / weight
            #print(1,means_matrix[i])
            # Subtracting the mean value from data
            scaled_input_diff = scaled_input - means_matrix[i]

            # Update the covariance matrix
            covariance_matrix[i] =
            (posterior_probability[:, i] *
            scaled_input_diff.T @ scaled_input_diff) /
            weight

            # Update the weights matrix
            weights_matrix[i] = weight / number_of_rows


        likelihood=0
        for i in range(k):
            try:
                pseudo_inverse =
                np.linalg.pinv(covariance_matrix[i] +
                np.diag(np.ones(covariance_matrix[i].shape[0]) * 1e-10))
                likelihood=likelihood+weights_matrix[i]*m
                ultivariate_normal.logpdf(scaled_input,me
                ans_matrix[i], sudo_inverse)
            except Exception as e:
                continue
        log_likelihood_new =np.sum(likelihood)

        new_means_matrix_df=pd.DataFrame(means_matrix)
        distance = []
        for col in new_means_matrix_df.columns:
            col_distance =
            euclidean(old_means_matrix_df[col],
            new_means_matrix_df[col])
            distance.append(col_distance)
        tao_calculated=sum(distance)/k
```

Problem 2 continued on next page. . .

```
        if tao_calculated<
        tao:#log_likelihood_new>log_likelihood_old and
        100*((log_likelihood_new - log_likelihood_old) /
        log_likelihood_old)<tao:

            print("Converged")
            labels=np.argmax(posterior_probability,axis=1
            )
            labels=maintain_k_clusters(labels,k)
            labels_df=pd.DataFrame(labels)
            means_matrix_df=pd.DataFrame(means_matrix,col
            umns=scaled_input_df.columns)
            sse=sum_of_square_error_em(means_matrix_df,
            scaled_input_df, labels_df)
            clainski=
            Calinski_index_em(scaled_input_df,labels_df)
            return sse,clainski,means_matrix_initial
        #else:
            #log_likelihood_old=log_likelihood_new

        if iterations>100:
            print("Max iteration reached")
            labels=np.argmax(posterior_probability,axis=1
            )
            labels=maintain_k_clusters(labels,k)
            labels_df=pd.DataFrame(labels)
            means_matrix_df=pd.DataFrame(means_matrix,col
            umns=scaled_input_df.columns)
            sse=sum_of_square_error_em(means_matrix_df,
            scaled_input_df, labels_df)
            clainski=
            Calinski_index_em(scaled_input_df,labels_df)
            return sse,clainski,means_matrix_initial


import time
from scipy.spatial.distance import euclidean
def initialize_centroids(df, k,means_matrix):
    """
    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=pd.DataFrame(means_matrix,columns=df.column
    s)
    centroids=centroids.T
    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 error_clusters(df_new_centroids,df1,df_label):
    """
    Calculate the error rate of each cluster.

    Args:
    - df_label (pandas.DataFrame): the label of the
    nearest centroid for each data point.
    - df1 (pandas.DataFrame): the dataset.
```

Problem 2 continued on next page. . .

```
    - df_new_centroids (pandas.DataFrame): The new centroids computed in the current iteration.

    Returns:
    - error_rate (float): the total error rate of all clusters.
    """


    #Calculate mean value
    mean_centroid=df1.groupby('readmitted').mean().reset_index()
    # Transpose the new centroids dataframe and reset the index
    new_centroids= df_new_centroids.T
    # Get the columns of the data dataframe
    columns = df1.columns

    sse = []
    # Compute the distance between each data point and its assigned centroid
    for i in range(len(new_centroids)):    #### centroid
        s=[]
        for j in range(len(mean_centroid)): ### mean centroid
        # Compute the distance between each data point and its assigned centroid
            distance =
            np.sum(np.square(mean_centroid[mean_centroid[
            'readmitted']==j][columns] -
            new_centroids.iloc[i][columns]), axis=1)
            s.append(distance.iloc[0])
        sse.append(s)
    ## key  is the cluster number and value is the merged value
    merge_label=pd.DataFrame(sse).idxmin(axis=1).to_dict()
    ## Merging cluster based on the target variable
    df_label[0]=df_label[0].replace(merge_label)

    df1 = df1.join(df_label) # add the label column to the dataset
    df1.rename(columns={0: 'Label'}, inplace=True) # rename the label column
    error_list = []
    for i in df1['Label'].value_counts().index:
        df_cluster = df1[df1['Label'] == i] # filter the
        dataset to include only the data points in the
        current cluster
        y = len(df_cluster[df_cluster['readmitted'] ==
        1]) # count the number of data points in the
        current cluster that were readmitted
        n = len(df_cluster[df_cluster['readmitted'] ==
        0]) # count the number of data points in the
        current cluster that were not readmitted
        if y == 0 and n == 0:
            error = 0
        else:
            error = n / (n + y) # calculate the error
            rate of the current cluster
        error_list.append(error)
    return round(sum(error_list),4)
```

```
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 = new_centroids.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 Calinski_index(df_data,clusters):
    ch_score = calinski_harabasz_score(df_data, clusters)
    return ch_score




def kmeans_lyod_with_error(df1, k, tou,means_matrix_initial):
    """
    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,means_matrix_initial) # Initialize random centroids
    initial_list_of_columns = centroids.columns.to_list()
```

Problem 2 continued on next page. . .

```
    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()
        clainski= Calinski_index(df1,df_label)
        if iteration>100:
            print("Iteration exceeded")

            return sse,clainski
            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,clainski
            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



scaler = StandardScaler()
scaler.fit(df_cleaned_dia)
scaled_input=scaler.transform(df_cleaned_dia)

scaled_input_df= pd.DataFrame(scaled_input,columns=df_cleaned_dia.columns)


error_matrix_em=[]
error_matrix_kmeans=[]
```

Problem 2 continued on next page. . .

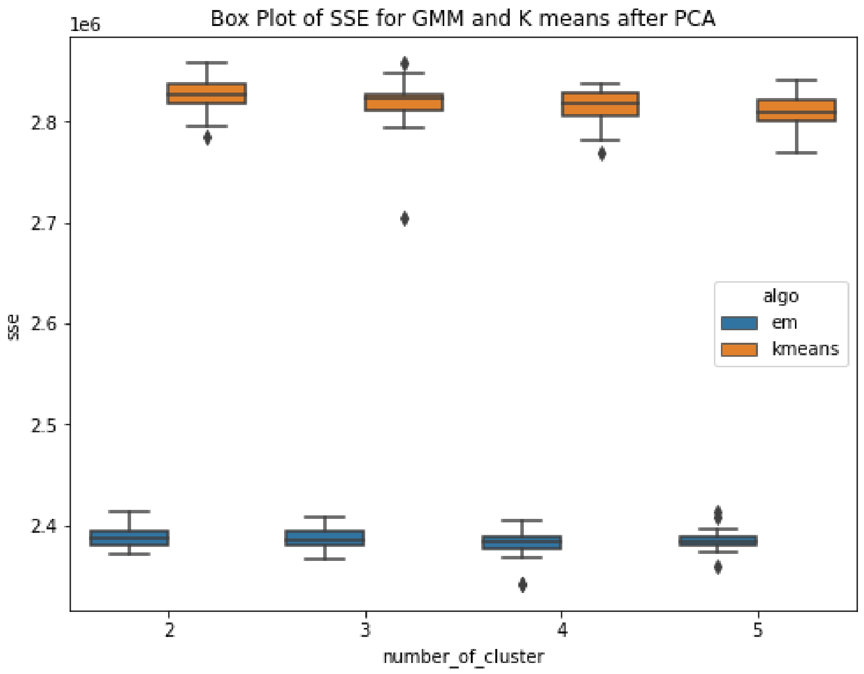
```
for i in range(2,6):
    for j in range(1,21):
        sse,clainski,means_matrix_initial=GMM(df_cleaned_
        dia,i,10)
        error_matrix_em.append([i,sse,clainski])

        sse,clainski=kmeans_lyod_with_error(scaled_input_
        df,i,10,means_matrix_initial)
        error_matrix_kmeans.append([i,sse,clainski])
error_df_em= pd.DataFrame(error_matrix_em,columns=['number_of_cluster', 'sse','clainski'])
error_df_kmeans= pd.DataFrame(error_matrix_kmeans,columns=
['number_of_cluster', 'sse','clainski'])


error_df_em.to_csv('6_em_pca.csv',index=False)
error_df_kmeans.to_csv('6_kmeans_pca.csv',index=False)


error_df_em['algo']='em'
error_df_kmeans['algo']='kmeans'


run_time_diab=pd.DataFrame()
run_time_diab=pd.concat( [ error_df_em[['algo','number_of_cluster','sse','clainski']],
    error_df_kmeans[['algo','number_of_cluster', 'sse','clainski']]
                        ],ignore_index=True )

import seaborn as sns

fig, ax = plt.subplots(figsize=(8,6))

sns.boxplot(x='number_of_cluster', y='sse', hue='algo',
            data=run_time_diab[run_time_diab['algo'].isin (['kmeans','em'])],ax=ax);
plt.title('Box Plot of SSE for GMM and K means after PCA')
plt.show()

import seaborn as sns

fig, ax = plt.subplots(figsize=(8,6))

sns.boxplot(x='number_of_cluster', y='clainski', hue='algo',
            data=run_time_diab[run_time_diab['algo'].isin (['kmeans','em'])],ax=ax);
plt.title('Box Plot of Clainski Index for GMM and K means after PCA')
plt.show()

error_df_em_nopca=pd.read_csv('6_em.csv')
error_df_kmeans_nopca=pd.read_csv('6_kmeans.csv')


error_df_em_nopca['algo']='em_nopca'
error_df_kmeans_nopca['algo']='kmeans_nopca'
```

Problem 2 continued on next page. . .

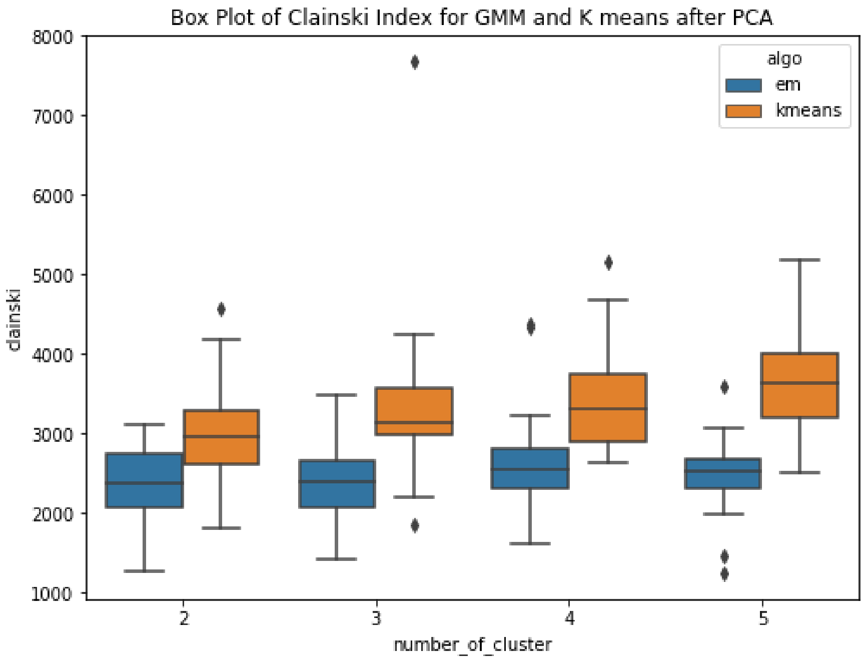
```
run_time_diab=pd.concat( [ run_time_diab[['algo','number_of_cluster','sse','clainski']],
    error_df_em_nopca[['algo','number_of_cluster', 'sse','clainski']],
    error_df_kmeans_nopca[['algo','number_of_cluster', 'sse','clainski']]
                        ],ignore_index=True )

import seaborn as sns

fig, ax = plt.subplots(figsize=(8,6))

sns.boxplot(x='number_of_cluster', y='sse', hue='algo',
            data=run_time_diab[run_time_diab['algo'].isin (['kmeans','em','kmeans_nopca','em_nop
plt.title('Box Plot of SSE for GMM and K means before and after PCA')
plt.show()


import seaborn as sns

fig, ax = plt.subplots(figsize=(8,6))

sns.boxplot(x='number_of_cluster', y='clainski', hue='algo',
            data=run_time_diab[run_time_diab['algo'].isin (['kmeans','em','kmeans_nopca','em_nop


plt.title('Box Plot of Clainski for GMM and K means before and after PCA')
plt.show()
```

Plot/s

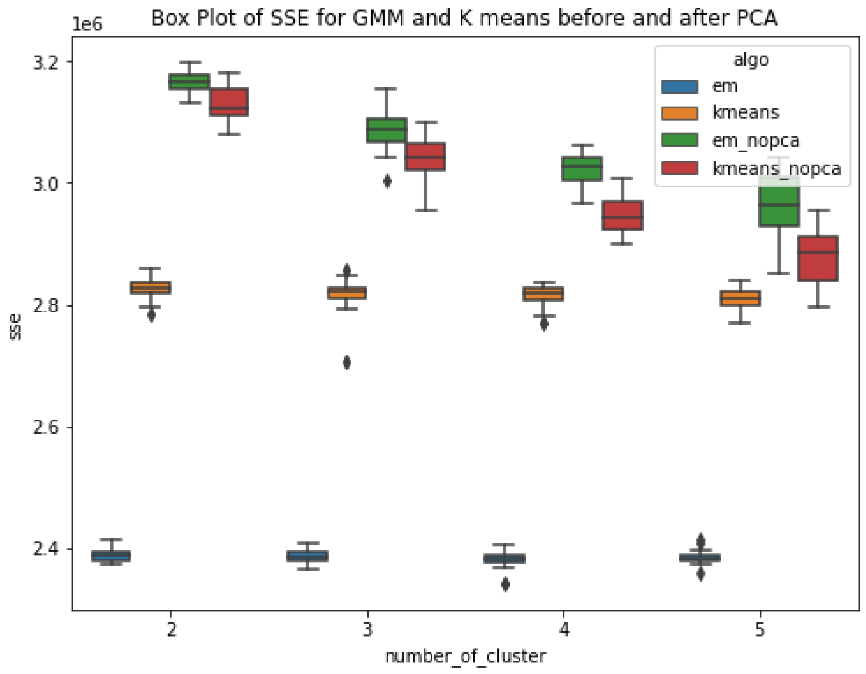
Place images here with suitable captions.



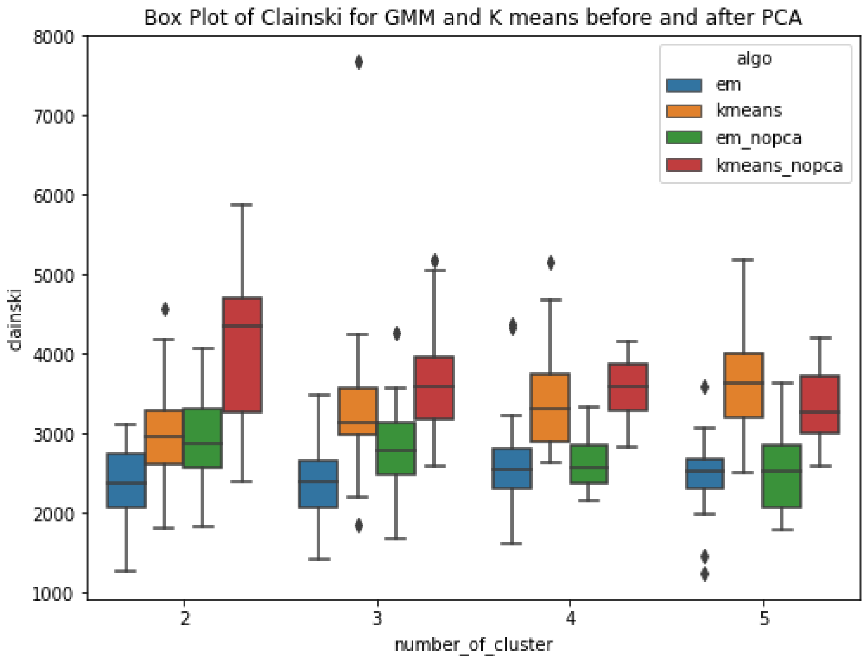
(7)


Box Plot of Clainski Index for GMM and K means after PCA
algo
em
kmeans
clainski
number_of_cluster
1000
2000
3000
4000
5000
6000
7000
8000
2
3
4
5

(8)


Box Plot of SSE for GMM and K means before and after PCA
1e6
algo
em
kmeans
em_nopca
kmeans_nopca
sse
number_of_cluster
2.4
2.6
2.8
3.0
3.2
2
3
4
5

(9)



(10)

Discussion of Experiments

Answer here... Have performed PCA over the diabetes dataset. To cover 90 percent of the variance, 24 principal components are required. So the dataset here gets reduced and now we use these principal components as data to run our GMM (EM) and kmeans algorithm. Have ran the algorithms for k = 2,3,4,5 and for each cluster the experiment is performed 20 times. PCA (Principal Component Analysis) is a method used for reducing the dimensionality of a dataset by transforming it into a lower-dimensional space while preserving most of the data variability. The transformed features resulting from PCA are linear combinations of the original features that are orthogonal to each other. PCA can help approximate a normal distribution of the data by removing redundant and correlated features and reducing the effects of outliers and noise.The aim is to improve the results. This is confirmed by the box plots. The box plot showing the within SSE between Kmeans and GMM shows the performance of GMM is much better than that of Kmeans after PCA. The median with sse is less for GMM as compared to that of the kmeans. The second plot shows the CHS score box plot for GMM and kmeans, shwoing the median value of CHS for kmeans is higher than that of the GMM. The third plot shows the comparision of within SSE for GMM and Kmeans before and after PCA. The plot shows that the sse after performing pca has much less value as compared to the previous experiments when PCA was not performed. The plot shows a huge difference in values of within sse for GMM with and without PCA and for kmeans. The fourth plot is a similar comparision but usinh CHS score as the evaluating metric. This shows that CHS score is higher before performing pca and reduces after performing PCA.

Problem 2 continued on next page...

4. Run the EM algorithm for the other two different mixture models such as, Poisson. Compare all your three G_k's using two different 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 [30 points].

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
from sklearn.preprocessing import StandardScaler
from scipy.stats import multivariate_normal
from sklearn.metrics import silhouette_score,
calinski_harabasz_score
from scipy.spatial.distance import euclidean

df= pd.read_csv('dataset_diabetes/diabetic_data.csv')


from sklearn.metrics import silhouette_score,
calinski_harabasz_score
def GMM_initialization(df,k):
    number_of_rows=df.shape[0]
    number_of_columns=df.shape[1]
    means_matrix = df.sample(n=k).values
    identity_matrix=np.eye(number_of_columns)
    covariance_matrix=np.array([identity_matrix]*k)
    weights_matrix=np.array([float(1/k)]*k)
    return
    means_matrix,covariance_matrix,weights_matrix,number_
    of_rows,number_of_columns


def calculate_posterior(data,means_matrix,covariance_matrix,w
eights_matrix,k,number_of_columns):
    posterior=np.zeros(k)

    gamma_arr = np.array([math.gamma(i+1) for i in data])
    for i in range(k):
        pois = (np.exp(-means_matrix[i]) * means_matrix[i] ** data)/gamma_arr
        posterior[i] = weights_matrix[i]*np.prod(pois)+0.0001

    return posterior

def maintain_k_clusters(labels,k):
    unique_values, value_counts = np.unique(labels,
    return_counts=True)
    missing_labels=[i for i in range(k) if i not in
    unique_values]
    unique_values_to_be_replaced=[unique_values[i] for i in np.where(value_counts > 1)[0]]

    indices=[i for i in range(len(labels))    if
    labels[i] in unique_values_to_be_replaced]
    random_indices = np.random.choice(indices,
    size=len(missing_labels), replace=False)
    for i,val in enumerate(random_indices):
        labels[val]=missing_labels[i]
    return labels


def sum_of_square_error_em(new_centroids, data, labels):
    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],dtype=np.float64), axis=1)
        #print(distance)
        sse.append(distance.sum())
    # Return the sum of squared errors

    a=sum(sse)
    return a

def Calinski_index_em(df_data,clusters):
    ch_score = calinski_harabasz_score(df_data, clusters)
    return ch_score



def GMM(df_cleaned_dia,k,tao):
    scaler = StandardScaler()
    scaler.fit(df_cleaned_dia)
    scaled_input=scaler.transform(df_cleaned_dia)

    scaled_input_df= pd.DataFrame(scaled_input,columns=df_cleaned_dia.colu
    mns)

    means_matrix,covariance_matrix,weights_matrix,number_of_rows,number_of_columns=
    GMM_initialization(scaled_input_df,k)
    likelihood=0
    means_matrix_initial=means_matrix
    for i in range(k):
        try:

            pseudo_inverse =
            np.linalg.pinv(covariance_matrix[i] + np.diag(np.ones(covariance_matrix[i].shape[0]
            )* 1e-10))
            likelihood=likelihood+weights_matrix[i]*multi
        variate_normal.logpdf(scaled_input,means_matrix
            [i], pseudo_inverse)
        except Exception as e:
            continue
    log_likelihood_old=np.sum(likelihood)
    old_means_matrix_df=pd.DataFrame(means_matrix)
    posterior_probability = np.zeros((scaled_input.shape[0], k))
    iterations=0

    while (True):
        iterations+=1
        # Expectation
        for i in range(scaled_input.shape[0]):
            posterior_probability[i] =
            calculate_posterior(scaled_input[i],
            means_matrix,covariance_matrix,weights_matrix,
            k,number_of_columns)

        # Maximization
        posterior_probability=np.nan_to_num(posterior_prob
        ability, nan=0)
        for i in range(k):
            # Calculating weight
            weight = posterior_probability[:, i].sum()
            #print(weight)
            # Updating each centroid
            means_matrix[i] = (posterior_probability[:,
            i] @ scaled_input) / weight
            #print(1,means_matrix[i])
            # Subtracting the mean value from data
            scaled_input_diff = scaled_input - means_matrix[i]

            # Update the covariance matrix
            covariance_matrix[i] =
            (posterior_probability[:, i] *
            scaled_input_diff.T @ scaled_input_diff) /
            weight

            # Update the weights matrix
            weights_matrix[i] = weight / number_of_rows


        likelihood=0
        for i in range(k):
            try:
                pseudo_inverse =
                np.linalg.pinv(covariance_matrix[i] +
                np.diag(np.ones(covariance_matrix[i].shape[0]) * 1e-10))
                likelihood=likelihood+weights_matrix[i]*m
                ultivariate_normal.logpdf(scaled_input,me
                ans_matrix[i], sudo_inverse)
            except Exception as e:
                continue
        log_likelihood_new =np.sum(likelihood)

        new_means_matrix_df=pd.DataFrame(means_matrix)
        distance = []
        for col in new_means_matrix_df.columns:
            col_distance =
            euclidean(old_means_matrix_df[col],
            new_means_matrix_df[col])
            distance.append(col_distance)
        tao_calculated=sum(distance)/k



        if tao_calculated<
        tao:#log_likelihood_new>log_likelihood_old and
        100*((log_likelihood_new - log_likelihood_old) /
        log_likelihood_old)<tao:

            print("Converged")
            labels=np.argmax(posterior_probability,axis=1
            )
            labels=maintain_k_clusters(labels,k)
            labels_df=pd.DataFrame(labels)
            means_matrix_df=pd.DataFrame(means_matrix,col
            umns=scaled_input_df.columns)
            sse=sum_of_square_error_em(means_matrix_df,
            scaled_input_df, labels_df)
            clainski=
            Calinski_index_em(scaled_input_df,labels_df)
            return sse,clainski,means_matrix_initial
        #else:
            #log_likelihood_old=log_likelihood_new

        if iterations>100:
            print("Max iteration reached")
            labels=np.argmax(posterior_probability,axis=1
            )
            labels=maintain_k_clusters(labels,k)
            labels_df=pd.DataFrame(labels)
            means_matrix_df=pd.DataFrame(means_matrix,col
            umns=scaled_input_df.columns)
            sse=sum_of_square_error_em(means_matrix_df,
            scaled_input_df, labels_df)
            clainski=
```

Problem 2 continued on next page. . .

```
            Calinski_index_em(scaled_input_df,labels_df)
            return sse,clainski,means_matrix_initial


error_matrix_em=[]

for i in range(2,6):
    for j in range(1,21):
        sse,clainski,means_matrix_initial=GMM(df_cleaned_
        dia,i,10)
        error_matrix_em.append([i,sse,clainski])

error_df_em= pd.DataFrame(error_matrix_em,columns=['number_of_cluster', 'sse','clainski'])

error_df_em.to_csv('6_em_poisson.csv',index=False)

import pandas as pd
import swifter
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import LabelEncoder
from tqdm import tqdm
from sklearn.preprocessing import StandardScaler
from scipy.stats import multivariate_normal
from sklearn.metrics import silhouette_score,
calinski_harabasz_score
from scipy.spatial.distance import euclidean

df= pd.read_csv('dataset_diabetes/diabetic_data.csv')


from sklearn.metrics import silhouette_score,
calinski_harabasz_score
def GMM_initialization(df,k):
    number_of_rows=df.shape[0]
    number_of_columns=df.shape[1]
    means_matrix = df.sample(n=k).values
    identity_matrix=np.eye(number_of_columns)
    covariance_matrix=np.array([identity_matrix]*k)
    weights_matrix=np.array([float(1/k)]*k)
    return
    means_matrix,covariance_matrix,weights_matrix,number_
    of_rows,number_of_columns



def calculate_posterior(data,means_matrix,covariance_matrix,w
eights_matrix,k,number_of_columns):
    posterior=np.zeros(k)

    for i in range(k):
        m=1/(means_matrix[i]+0.01)
        exp= np.exp(-m*data)/m
        posterior[i] = weights_matrix[i]*np.prod(exp)+0.0001
    return posterior


def maintain_k_clusters(labels,k):
    unique_values, value_counts = np.unique(labels,
    return_counts=True)
    missing_labels=[i for i in range(k) if i not in
    unique_values]
    unique_values_to_be_replaced=[unique_values[i] for i in np.where(value_counts > 1)[0]]

    indices=[i for i in range(len(labels))    if
    labels[i] in unique_values_to_be_replaced]
    random_indices = np.random.choice(indices,
    size=len(missing_labels), replace=False)
    for i,val in enumerate(random_indices):
        labels[val]=missing_labels[i]
    return labels


def sum_of_square_error_em(new_centroids, data, labels):
    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],dtype=np.float64), axis=1)
        #print(distance)
        sse.append(distance.sum())
    # Return the sum of squared errors

    a=sum(sse)
    return a

def Calinski_index_em(df_data,clusters):
    ch_score = calinski_harabasz_score(df_data, clusters)
    return ch_score



def GMM(df_cleaned_dia,k,tao):
    scaler = StandardScaler()
    scaler.fit(df_cleaned_dia)
    scaled_input=scaler.transform(df_cleaned_dia)
```

Problem 2 continued on next page. . .

```
    scaled_input_df= pd.DataFrame(scaled_input,columns=df_cleaned_dia.colu
    mns)

    means_matrix,covariance_matrix,weights_matrix,number_of_rows,number_of_columns=
    GMM_initialization(scaled_input_df,k)
    likelihood=0
    means_matrix_initial=means_matrix
    for i in range(k):
        try:

            pseudo_inverse =
            np.linalg.pinv(covariance_matrix[i] + np.diag(np.ones(covariance_matrix[i].shape[0]
            )* 1e-10))
            likelihood=likelihood+weights_matrix[i]*multi
        variate_normal.logpdf(scaled_input,means_matrix
            [i], pseudo_inverse)
        except Exception as e:
            continue
    log_likelihood_old=np.sum(likelihood)
    old_means_matrix_df=pd.DataFrame(means_matrix)
    posterior_probability = np.zeros((scaled_input.shape[0], k))
    iterations=0

    while (True):
        iterations+=1
        # Expectation
        for i in range(scaled_input.shape[0]):
            posterior_probability[i] =
            calculate_posterior(scaled_input[i],
            means_matrix,covariance_matrix,weights_matrix,
            k,number_of_columns)

        # Maximization
        posterior_probability=np.nan_to_num(posterior_prob
        ability, nan=0)
        for i in range(k):
            # Calculating weight
            weight = posterior_probability[:, i].sum()
            #print(weight)
            # Updating each centroid
            means_matrix[i] = (posterior_probability[:,
            i] @ scaled_input) / weight
            #print(1,means_matrix[i])
            # Subtracting the mean value from data
            scaled_input_diff = scaled_input - means_matrix[i]

            # Update the covariance matrix
            covariance_matrix[i] =
            (posterior_probability[:, i] *
            scaled_input_diff.T @ scaled_input_diff) /
            weight

            # Update the weights matrix
            weights_matrix[i] = weight / number_of_rows


        likelihood=0
        for i in range(k):
            try:
                pseudo_inverse =
                np.linalg.pinv(covariance_matrix[i] +
                np.diag(np.ones(covariance_matrix[i].shape[0]) * 1e-10))
                likelihood=likelihood+weights_matrix[i]*m
                ultivariate_normal.logpdf(scaled_input,me
                ans_matrix[i], sudo_inverse)
            except Exception as e:
                continue
        log_likelihood_new =np.sum(likelihood)

        new_means_matrix_df=pd.DataFrame(means_matrix)
        distance = []
        for col in new_means_matrix_df.columns:
            col_distance =
            euclidean(old_means_matrix_df[col],
            new_means_matrix_df[col])
            distance.append(col_distance)
        tao_calculated=sum(distance)/k



        if tao_calculated<
        tao:#log_likelihood_new>log_likelihood_old and
        100*((log_likelihood_new - log_likelihood_old) /
        log_likelihood_old)<tao:

            print("Converged")
            labels=np.argmax(posterior_probability,axis=1
            )
            labels=maintain_k_clusters(labels,k)
            labels_df=pd.DataFrame(labels)
            means_matrix_df=pd.DataFrame(means_matrix,col
            umns=scaled_input_df.columns)
            sse=sum_of_square_error_em(means_matrix_df,
            scaled_input_df, labels_df)
            clainski=
            Calinski_index_em(scaled_input_df,labels_df)
            return sse,clainski,means_matrix_initial
        #else:
            #log_likelihood_old=log_likelihood_new

        if iterations>100:
            print("Max iteration reached")
            labels=np.argmax(posterior_probability,axis=1
            )
            labels=maintain_k_clusters(labels,k)
            labels_df=pd.DataFrame(labels)
            means_matrix_df=pd.DataFrame(means_matrix,col
            umns=scaled_input_df.columns)
            sse=sum_of_square_error_em(means_matrix_df,
            scaled_input_df, labels_df)
            clainski=
            Calinski_index_em(scaled_input_df,labels_df)
            return sse,clainski,means_matrix_initial


error_matrix_em=[]

error_matrix_em=[]

for i in range(2,6):
    for j in range(1,21):
        sse,clainski,means_matrix_initial=GMM(df_cleaned_dia,i,10)
        error_matrix_em.append([i,sse,clainski])

error_df_em= pd.DataFrame(error_matrix_em,columns=['number_of_cluster', 'sse','clainski'])

error_df_em.to_csv('6_em_exp.csv',index=False)



from sklearn.metrics import silhouette_score,
calinski_harabasz_score
def GMM_initialization(df,k):
    number_of_rows=df.shape[0]
    number_of_columns=df.shape[1]
    means_matrix = df.sample(n=k).values
    identity_matrix=np.eye(number_of_columns)
    covariance_matrix=np.array([identity_matrix]*k)
    weights_matrix=np.array([float(1/k)]*k)
    return
    means_matrix,covariance_matrix,weights_matrix,number_
    of_rows,number_of_columns


def
calculate_posterior(data,means_matrix,covariance_matrix,w
eights_matrix,k,number_of_columns):
    posterior=np.zeros(k)
    for i in range(k):
        try:
            pseudo_inverse =
            np.linalg.pinv(covariance_matrix[i] +
            np.eye(covariance_matrix[i].shape[0]) * 1e-
            2,
            rcond=1e-10)
            posterior[i] = multivariate_normal.pdf(data,
            mean=means_matrix[i], cov=pseudo_inverse)
        except Exception as e:
            continue
    return posterior*weights_matrix/(posterior*weights_matrix).sum()

def maintain_k_clusters(labels,k):
    unique_values, value_counts = np.unique(labels,
    return_counts=True)
    missing_labels=[i for i in range(k) if i not in
    unique_values]
    unique_values_to_be_replaced=[unique_values[i] for i in np.where(value_counts > 1)[0]]

    indices=[i for i in range(len(labels))    if
    labels[i] in unique_values_to_be_replaced]
    random_indices = np.random.choice(indices,
    size=len(missing_labels), replace=False)
    for i,val in enumerate(random_indices):
        labels[val]=missing_labels[i]
    return labels


def sum_of_square_error_em(new_centroids, data, labels):
    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],dtype=np.float64), axis=1)
        #print(distance)
        sse.append(distance.sum())
    # Return the sum of squared errors

    a=sum(sse)
    return a

def Calinski_index_em(df_data,clusters):
    ch_score = calinski_harabasz_score(df_data, clusters)
    return ch_score



def GMM(df_cleaned_dia,k,tao):
    scaler = MinMaxScaler()
    scaler.fit(df_cleaned_dia)
    scaled_input=scaler.transform(df_cleaned_dia)

    scaled_input_df= pd.DataFrame(scaled_input,columns=df_cleaned_dia.colu
    mns)
```

Problem 2 continued on next page. . .

```
    means_matrix,covariance_matrix,weights_matrix,number_of_rows,number_of_columns=
    GMM_initialization(scaled_input_df,k)
    likelihood=0
    means_matrix_initial=means_matrix
    for i in range(k):
        try:

            pseudo_inverse =
            np.linalg.pinv(covariance_matrix[i] + np.diag(np.ones(covariance_matrix[i].shape[0]
            )* 1e-10))
            likelihood=likelihood+weights_matrix[i]*multi
        variate_normal.logpdf(scaled_input,means_matrix
            [i], pseudo_inverse)
        except Exception as e:
            continue
    log_likelihood_old=np.sum(likelihood)
    old_means_matrix_df=pd.DataFrame(means_matrix)
    posterior_probability = np.zeros((scaled_input.shape[0], k))
    iterations=0

    while (True):
        iterations+=1
        # Expectation
        for i in range(scaled_input.shape[0]):
            posterior_probability[i] =
            calculate_posterior(scaled_input[i],
            means_matrix,covariance_matrix,weights_matrix,
            k,number_of_columns)

        # Maximization
        posterior_probability=np.nan_to_num(posterior_prob
        ability, nan=0)
        for i in range(k):
            # Calculating weight
            weight = posterior_probability[:, i].sum()
            #print(weight)
            # Updating each centroid
            means_matrix[i] = (posterior_probability[:,
            i] @ scaled_input) / weight
            #print(1,means_matrix[i])
            # Subtracting the mean value from data
            scaled_input_diff = scaled_input - means_matrix[i]

            # Update the covariance matrix
            covariance_matrix[i] =
            (posterior_probability[:, i] *
            scaled_input_diff.T @ scaled_input_diff) /
            weight

            # Update the weights matrix
            weights_matrix[i] = weight / number_of_rows
```

Problem 2 continued on next page. . .

```
        likelihood=0
        for i in range(k):
            try:
                pseudo_inverse =
                np.linalg.pinv(covariance_matrix[i] +
                np.diag(np.ones(covariance_matrix[i].shape[0]) * 1e-10))
                likelihood=likelihood+weights_matrix[i]*m
                ultivariate_normal.logpdf(scaled_input,me
                ans_matrix[i], sudo_inverse)
            except Exception as e:
                continue
        log_likelihood_new =np.sum(likelihood)

        new_means_matrix_df=pd.DataFrame(means_matrix)
        distance = []
        for col in new_means_matrix_df.columns:
            col_distance =
            euclidean(old_means_matrix_df[col],
            new_means_matrix_df[col])
            distance.append(col_distance)
        tao_calculated=sum(distance)/k



        if tao_calculated<
        tao:#log_likelihood_new>log_likelihood_old and
        100*((log_likelihood_new - log_likelihood_old) /
        log_likelihood_old)<tao:

            print("Converged")
            labels=np.argmax(posterior_probability,axis=1
            )
            labels=maintain_k_clusters(labels,k)
            labels_df=pd.DataFrame(labels)
            means_matrix_df=pd.DataFrame(means_matrix,col
            umns=scaled_input_df.columns)
            sse=sum_of_square_error_em(means_matrix_df,
            scaled_input_df, labels_df)
            clainski=
            Calinski_index_em(scaled_input_df,labels_df)
            return sse,clainski,means_matrix_initial
        #else:
            #log_likelihood_old=log_likelihood_new

        if iterations>100:
            print("Max iteration reached")
            labels=np.argmax(posterior_probability,axis=1
            )
            labels=maintain_k_clusters(labels,k)
            labels_df=pd.DataFrame(labels)
            means_matrix_df=pd.DataFrame(means_matrix,col
            umns=scaled_input_df.columns)
```

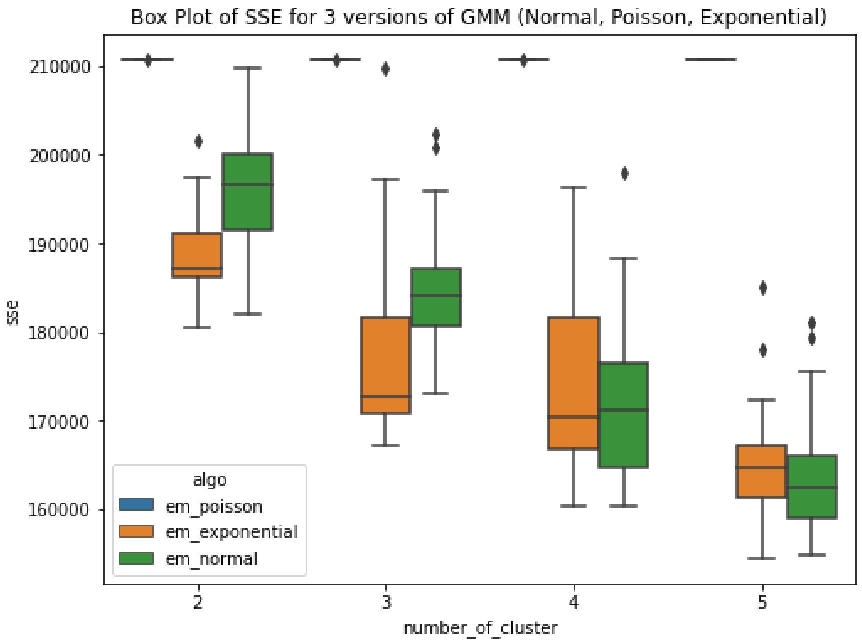
Problem 2 continued on next page. . .

```
            sse=sum_of_square_error_em(means_matrix_df,
            scaled_input_df, labels_df)
            clainski=
            Calinski_index_em(scaled_input_df,labels_df)
            return sse,clainski,means_matrix_initial


error_df_em.to_csv('6_em_minmax.csv',index=False)


error_df_em_poisson=pd.read_csv('6_em_poisson.csv')
error_df_em_exp=pd.read_csv('6_em_exp.csv')
error_df_em_normal=pd.read_csv('6_em_minmax.csv')

error_df_em_poisson['algo']='em_poisson'
error_df_em_exp['algo']='em_exponential'
error_df_em_normal['algo']='em_normal'

run_time_diab=pd.DataFrame()
run_time_diab=pd.concat( [ error_df_em_poisson[['algo','number_of_cluster','sse','clainski']],
    error_df_em_exp[['algo','number_of_cluster', 'sse','clainski']],
     error_df_em_normal[['algo','number_of_cluster', 'sse','clainski']]
                        ],ignore_index=True )

import seaborn as sns

fig, ax = plt.subplots(figsize=(8,6))

sns.boxplot(x='number_of_cluster', y='sse', hue='algo',
            data=run_time_diab[run_time_diab['algo'].isin (['em_poisson','em_exponential','em_nc
plt.title('Box Plot of SSE for 3 versions of GMM (Normal, Poisson, Exponential)')
plt.show()


import seaborn as sns

fig, ax = plt.subplots(figsize=(8,6))

sns.boxplot(x='number_of_cluster', y='clainski', hue='algo',
            data=run_time_diab[run_time_diab['algo'].isin (['em_poisson','em_exponential','em_nc
plt.title('Box Plot of Clainski for 3 versions of GMM (Normal, Poisson, Exponential)')
plt.show()
```

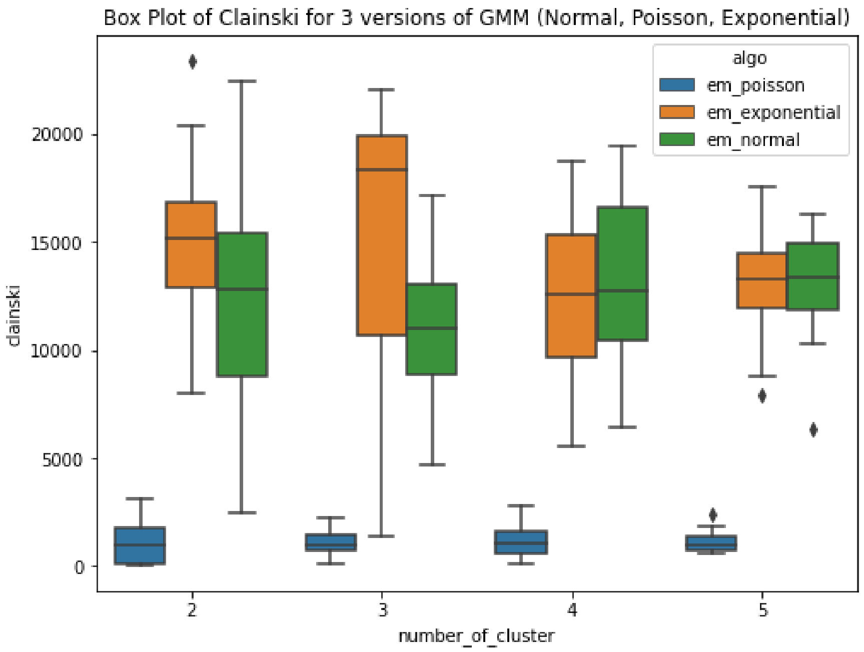
Problem 2 continued on next page. . .

Plot/s

Place images here with suitable captions.



(11)



(12)

Discussion of Experiments

Answer here. . . In this question, other than normal distribution, I have tried the following distributions:
1) Poisson Distribution

2) Exponential Distribution Here the Poisson distribution requires positive values for data, so have used MinMaxScaler to scale the data and then have created 3 different codes, one for each distribution type i.e normal, Poisson and exponential. Equation for Poisson Distribution P(X=k) = $\lambda^k e^{-\lambda} \overline{k!}$

Equation for Expoenential Distribution f(x; λ) = $\lambda e^{-\lambda x}$

From the box plot of within sum of sqaure error , it can be seen that with increase in the number of clusters, the within SSE of normal distribution improves and for k=5 it is the best. The within sse for poisson is highest. The reason could be the data is normally distributed and hence the assumption that data is normally distributed is correct. Also, in previous assignments EDA, it was identified the data was normally distributed and was skewed. Similarly, the CHS score shows that with increase in number of cluster, the score improves and for k=4 and 5 the normal distribution outperforms other distributions(poisson and exponential).

Problem 3

Improve the EM algorithm through initialization. *k*-means ++ is an extended *k*-means clustering algorithm and induces non-uniform distributions over the data that serve as the initial centroids. Read the paper and implement this idea to improve your G_k program. Let's call the new algorithm G_{k++}. Run your new G_{k++} and G_k for $k = 2, \ldots, 5$ for 20 runs each. Compare G_k and C_k using two different 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.

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
from sklearn.preprocessing import StandardScaler
from scipy.stats import multivariate_normal
from sklearn.metrics import silhouette_score, calinski_harabasz_score
from scipy.spatial.distance import euclidean

df= pd.read_csv('dataset_diabetes/diabetic_data.csv')

def initialize_centroids_plus(df,k):
    """
    Function to calculate the random centroid using
    kmeans ++ technique.
    Input:
        - df: pandas dataframe with the data
        - k: number of clusters
    Output:
        - centroid.T: pandas dataframe with all
        centroids initialized
    """
    #Initialize random centroids from dataset
    centroid = []
    centroid.append(df.apply(lambda x:
    float(x.sample())))
    centroid=pd.DataFrame(centroid)
    column=centroid.columns.to_list()
    ##Randomly created first centroid from the domain of
    each column in the dataframe

    for i in range(1,k):
    ## The above for loop is for generating k-1 clusters
    as first random cluster is already generated
        distance=pd.DataFrame()
    ## Creating dataframe of distance. This will store
    the distance of each datapoint from each cluster
        for j in range(len(centroid)):
    ##This for loop is for finding distance from each
    centroid
            a=pd.DataFrame([np.sqrt(np.sum(np.square(df[c
            olumn] - centroid.iloc[j][column]),
            axis=1))]).T

            distance=pd.concat([distance,a],axis=1)
    ## Distance_min stores the minimum distance of each point from all the centroids
        distance_min=distance.min(axis=1)
    ##Calculates probability for each datapoint
        probability = distance_min / distance_min.sum()
    ## Selecting the next centroid based on the
    probability which is proportional to find square
    distance
        new_centroid =
        pd.DataFrame(df.iloc[np.random.choice(len(df),p=p
        robability)]).T

        centroid=pd.concat([centroid,new_centroid],ignore
        _index=True)
        centroid.index.name='Label'
    ## Concatenated the centroid dataframe with new
    centroid and loop continues until all k centroids
    are initialized randomly
    return centroid


from sklearn.metrics import silhouette_score,
calinski_harabasz_score
def GMM_initialization(df,k):
    number_of_rows=df.shape[0]
    number_of_columns=df.shape[1]
    a = initialize_centroids_plus(df,k)
    means_matrix= a.to_numpy()
    identity_matrix=np.eye(number_of_columns)
    covariance_matrix=np.array([identity_matrix]*k)
    weights_matrix=np.array([float(1/k)]*k)
    return
    means_matrix,covariance_matrix,weights_matrix,number_
    of_rows,number_of_columns


def
calculate_posterior(data,means_matrix,covariance_matrix,w
eights_matrix,k,number_of_columns):
    posterior=np.zeros(k)
    for i in range(k):
        try:
            pseudo_inverse =
            np.linalg.pinv(covariance_matrix[i] +
            np.eye(covariance_matrix[i].shape[0]) * 1e-
            2,
            rcond=1e-10)
            posterior[i] = multivariate_normal.pdf(data,
            mean=means_matrix[i], cov=pseudo_inverse)
        except Exception as e:
            continue
    return posterior*weights_matrix/(posterior*weights_matrix).sum()

def maintain_k_clusters(labels,k):
    unique_values, value_counts = np.unique(labels,
    return_counts=True)
    missing_labels=[i for i in range(k) if i not in
    unique_values]
    unique_values_to_be_replaced=[unique_values[i] for i in np.where(value_counts > 1)[0]]

    indices=[i for i in range(len(labels))    if
    labels[i] in unique_values_to_be_replaced]
    random_indices = np.random.choice(indices,
    size=len(missing_labels), replace=False)
    for i,val in enumerate(random_indices):
        labels[val]=missing_labels[i]
    return labels


def sum_of_square_error_em(new_centroids, data, labels):
    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],dtype=np.float64), axis=1)
        #print(distance)
        sse.append(distance.sum())
    # Return the sum of squared errors

    a=sum(sse)
    return a

def Calinski_index_em(df_data,clusters):
    ch_score = calinski_harabasz_score(df_data, clusters)
    return ch_score



def GMM(df_cleaned_dia,k,tao):
    scaler = StandardScaler()
    scaler.fit(df_cleaned_dia)
    scaled_input=scaler.transform(df_cleaned_dia)

    scaled_input_df= pd.DataFrame(scaled_input,columns=df_cleaned_dia.colu
    mns)

    means_matrix,covariance_matrix,weights_matrix,number_of_rows,number_of_columns=
    GMM_initialization(scaled_input_df,k)
    likelihood=0
    means_matrix_initial=means_matrix
    for i in range(k):
        try:

            pseudo_inverse =
            np.linalg.pinv(covariance_matrix[i] + np.diag(np.ones(covariance_matrix[i].shape[0]
            )* 1e-10))
            likelihood=likelihood+weights_matrix[i]*multi
        variate_normal.logpdf(scaled_input,means_matrix
            [i], pseudo_inverse)
        except Exception as e:
            continue
    log_likelihood_old=np.sum(likelihood)
    old_means_matrix_df=pd.DataFrame(means_matrix)
    posterior_probability = np.zeros((scaled_input.shape[0], k))
    iterations=0

    while (True):
        iterations+=1
        # Expectation
        for i in range(scaled_input.shape[0]):
            posterior_probability[i] =
            calculate_posterior(scaled_input[i],
            means_matrix,covariance_matrix,weights_matrix,
            k,number_of_columns)

        # Maximization
        posterior_probability=np.nan_to_num(posterior_prob
        ability, nan=0)
```

Problem 3 continued on next page. . .

```
        for i in range(k):
            # Calculating weight
            weight = posterior_probability[:, i].sum()
            #print(weight)
            # Updating each centroid
            means_matrix[i] = (posterior_probability[:,
            i] @ scaled_input) / weight
            #print(1,means_matrix[i])
            # Subtracting the mean value from data
            scaled_input_diff = scaled_input - means_matrix[i]

            # Update the covariance matrix
            covariance_matrix[i] =
            (posterior_probability[:, i] *
            scaled_input_diff.T @ scaled_input_diff) /
            weight

            # Update the weights matrix
            weights_matrix[i] = weight / number_of_rows


        likelihood=0
        for i in range(k):
            try:
                pseudo_inverse =
                np.linalg.pinv(covariance_matrix[i] +
                np.diag(np.ones(covariance_matrix[i].shape[0]) * 1e-10))
                likelihood=likelihood+weights_matrix[i]*m
                ultivariate_normal.logpdf(scaled_input,me
                ans_matrix[i], sudo_inverse)
            except Exception as e:
                continue
        log_likelihood_new =np.sum(likelihood)

        new_means_matrix_df=pd.DataFrame(means_matrix)
        distance = []
        for col in new_means_matrix_df.columns:
            col_distance =
            euclidean(old_means_matrix_df[col],
            new_means_matrix_df[col])
            distance.append(col_distance)
        tao_calculated=sum(distance)/k



        if tao_calculated<
        tao:#log_likelihood_new>log_likelihood_old and
        100*((log_likelihood_new - log_likelihood_old) /
        log_likelihood_old)<tao:

            print("Converged")
            labels=np.argmax(posterior_probability,axis=1
            )
            labels=maintain_k_clusters(labels,k)
            labels_df=pd.DataFrame(labels)
            means_matrix_df=pd.DataFrame(means_matrix,col
            umns=scaled_input_df.columns)
            sse=sum_of_square_error_em(means_matrix_df,
            scaled_input_df, labels_df)
            clainski=
            Calinski_index_em(scaled_input_df,labels_df)
            return sse,clainski,means_matrix_initial
        #else:
            #log_likelihood_old=log_likelihood_new

        if iterations>100:
            print("Max iteration reached")
            labels=np.argmax(posterior_probability,axis=1
            )
            labels=maintain_k_clusters(labels,k)
            labels_df=pd.DataFrame(labels)
            means_matrix_df=pd.DataFrame(means_matrix,col
            umns=scaled_input_df.columns)
            sse=sum_of_square_error_em(means_matrix_df,
            scaled_input_df, labels_df)
            clainski=
            Calinski_index_em(scaled_input_df,labels_df)
            return sse,clainski,means_matrix_initial


import time
from scipy.spatial.distance import euclidean
def initialize_centroids(df, k,means_matrix):
    """
    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=pd.DataFrame(means_matrix,columns=df.column
    s)
    centroids=centroids.T
    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
```

Problem 3 continued on next page. . .

```
        - 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 error_clusters(df_new_centroids,df1,df_label):
    """
    Calculate the error rate of each cluster.

    Args:
    - df_label (pandas.DataFrame): the label of the
    nearest centroid for each data point.
    - df1 (pandas.DataFrame): the dataset.
    - df_new_centroids (pandas.DataFrame): The new centroids computed in the current iteration.

    Returns:
    - error_rate (float): the total error rate of all clusters.
    """


    #Calculate mean value
    mean_centroid=df1.groupby('readmitted').mean().reset_index()
```

Problem 3 continued on next page. . .

```
    # Transpose the new centroids dataframe and reset the index
    new_centroids= df_new_centroids.T
    # Get the columns of the data dataframe
    columns = df1.columns

    sse = []
    # Compute the distance between each data point and its assigned centroid
    for i in range(len(new_centroids)):   #### centroid
        s=[]
        for j in range(len(mean_centroid)): ### mean centroid
        # Compute the distance between each data point and its assigned centroid
            distance =
            np.sum(np.square(mean_centroid[mean_centroid[
            'readmitted']==j][columns] -
            new_centroids.iloc[i][columns]), axis=1)
            s.append(distance.iloc[0])
        sse.append(s)
    ## key  is the cluster number and value is the merged value
    merge_label=pd.DataFrame(sse).idxmin(axis=1).to_dict()
    ## Merging cluster based on the target variable
    df_label[0]=df_label[0].replace(merge_label)

    df1 = df1.join(df_label) # add the label column to the dataset
    df1.rename(columns={0: 'Label'}, inplace=True) # rename the label column
    error_list = []
    for i in df1['Label'].value_counts().index:
        df_cluster = df1[df1['Label'] == i] # filter the
        dataset to include only the data points in the
        current cluster
        y = len(df_cluster[df_cluster['readmitted'] ==
        1]) # count the number of data points in the
        current cluster that were readmitted
        n = len(df_cluster[df_cluster['readmitted'] ==
        0]) # count the number of data points in the
        current cluster that were not readmitted
        if y == 0 and n == 0:
            error = 0
        else:
            error = n / (n + y) # calculate the error
            rate of the current cluster
        error_list.append(error)
    return round(sum(error_list),4)


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.
```

Problem 3 continued on next page. . .

```
    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 = new_centroids.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 Calinski_index(df_data,clusters):
    ch_score = calinski_harabasz_score(df_data, clusters)
    return ch_score




def kmeans_lyod_with_error(df1, k, tou,means_matrix_initial):
    """
    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,means_matrix_initial) # 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()
```

Problem 3 continued on next page. . .

```
        # 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()
        clainski= Calinski_index(df1,df_label)
        if iteration>100:
            print("Iteration exceeded")

            return sse,clainski
            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,clainski
            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



scaler = StandardScaler()
scaler.fit(df_cleaned_dia)
scaled_input=scaler.transform(df_cleaned_dia)

scaled_input_df= pd.DataFrame(scaled_input,columns=df_cleaned_dia.columns)


error_matrix_em=[]
error_matrix_kmeans=[]
for i in range(2,6):
    for j in range(1,21):
        sse,clainski,means_matrix_initial=GMM(df_cleaned_dia,i,10)
        error_matrix_em.append([i,sse,clainski])

        sse,clainski=kmeans_lyod_with_error(scaled_input_df,i,10,means_matrix_initial)
        error_matrix_kmeans.append([i,sse,clainski])
error_df_em= pd.DataFrame(error_matrix_em,columns=['number_of_cluster', 'sse','clainski'])
error_df_kmeans= pd.DataFrame(error_matrix_kmeans,columns=['number_of_cluster', 'sse','clainski'])
```

```
error_df_em.to_csv('6_em_++.csv',index=False)
error_df_kmeans.to_csv('6_kmeans_++.csv',index=False)

error_df_em.to_csv('6_em.csv',index=False)
error_df_kmeans.to_csv('6_kmeans.csv',index=False)

error_df_em_normal=pd.read_csv('6_em.csv')
error_df_kmeans_normal=pd.read_csv('6_kmeans.csv')


error_df_em['algo']='em++'
error_df_kmeans['algo']='kmeans++'
error_df_em_normal['algo']='em'
error_df_kmeans_normal['algo']='kmeans'

run_time_diab=pd.DataFrame()
run_time_diab=pd.concat( [ error_df_em[['algo','number_of_cluster','sse','clainski']],
    error_df_kmeans[['algo','number_of_cluster', 'sse','clainski']],
    error_df_em_normal[['algo','number_of_cluster', 'sse','clainski']],
    error_df_kmeans_normal[['algo','number_of_cluster', 'sse','clainski']]


                        ],ignore_index=True )

import seaborn as sns

fig, ax = plt.subplots(figsize=(8,6))

sns.boxplot(x='number_of_cluster', y='sse', hue='algo',
            data=run_time_diab[run_time_diab['algo'].isin (['em++','kmeans++'])],ax=ax);
plt.title('Box Plot of SSE for GMM++ and K means++ initialization')
plt.show()

import seaborn as sns

fig, ax = plt.subplots(figsize=(8,6))

sns.boxplot(x='number_of_cluster', y='clainski', hue='algo',
            data=run_time_diab[run_time_diab['algo'].isin (['em++','kmeans++'])],ax=ax);
plt.title('Box Plot of Clainski for GMM++ and K means++ initialization')
plt.show()

import seaborn as sns

fig, ax = plt.subplots(figsize=(8,6))

sns.boxplot(x='number_of_cluster', y='sse', hue='algo',
            data=run_time_diab[run_time_diab['algo'].isin (['kmeans','em','em++','kmeans++'])],ax=ax
plt.title('Box Plot of SSE for GMM and K means normal and with ++ initialization')
plt.show()

import seaborn as sns
```

```
fig, ax = plt.subplots(figsize=(8,6))

sns.boxplot(x='number_of_cluster', y='clainski', hue='algo',
            data=run_time_diab[run_time_diab['algo'].isin(['kmeans','em','em++','kmeans++'])],ax=ax)
plt.title('Box Plot of SSE for GMM and K means normal and with ++ initialization')
plt.show()
```

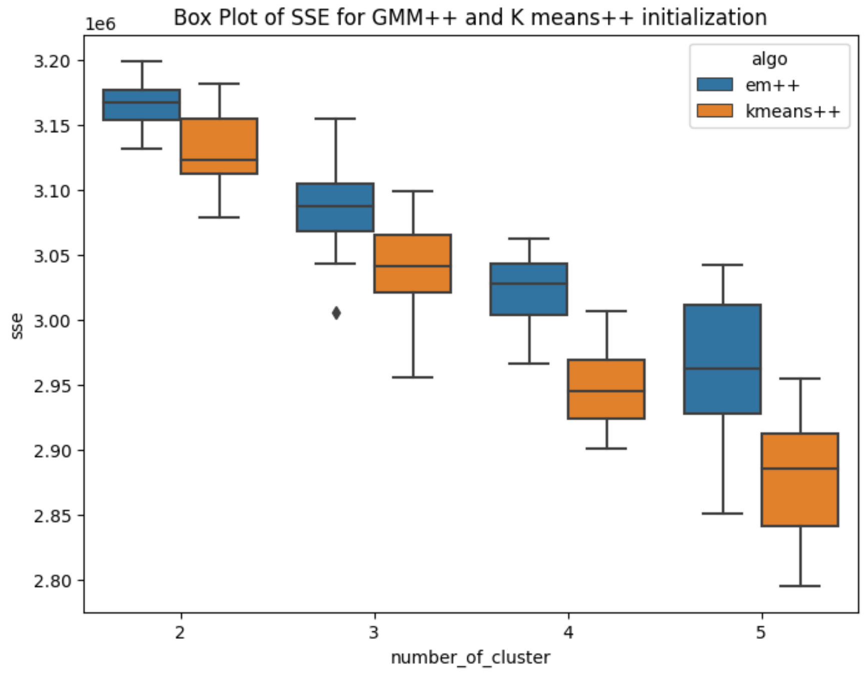
Discussion of Experiments

Answer here...
I have implemented the kmeans plus plus algorithm for number of clusters ranging from 2 to 5. The method is similar to kmeans llyod with different centroid initialization. Here the first centroid is chosen randomly from the domain of data. For rest k-1 centroids, we follow a different iterative approach. The second cluster is initialized based on the first centroid. The third centroid requires data of first two centroids. The methodolgy includes,selecting the first centroid randomly from domain of data. Then second cluster is at the farthest distance from that cluster. In this way iteratively we initialize k centroids. As the centroid initialization is not random, it is expected that the inter cluster distance will be more and intra cluster distance bewteen points and centroid will be less. The K-means++ algorithm selects the centroids in following way Step1: Choosing the first centroid at random from the data points. Step2: For each remaining data point, computing its distance to the nearest centroid that has already been chosen. Step3: Selecting the next centroid randomly from the remaining data points, with probability proportional to the squared distance to the nearest centroid. Repeating steps 2-3 until all K centroids have been chosen. The main idea behind K-means++ initialization is to select centroids that are well spread out across the data points. By selecting the next centroid from the remaining data points with a probability that is proportional to the squared distance to the nearest centroid, K-means++ initialization ensures that data points that are far away from existing centroids are more likely to be selected as new centroids. Hence the sum of square error is expected to be less. Rest the stopping conditions and other steps are similar to that of kmeans llyods.
Similarly the mean matrix is initialized for GMM(EM) and same is used for K means. HAve ran this experiment for k=2,3,4,5 , running 20 times for each cluster. The first box plot shows the within Sum of square error for GMM plus plus and kmeans plus plus. It can be seen that, the median within SSE of kmeans is less than that of GMM. The second plot shows the CHS score using the new initialization technique and the value of CHS score for kmeans is higher than that of GMM. The third and fourth box plot shows the comparision of sse and CHS score for normal GMM and kmeans with the GMM plus Plus and Kmeans plus plus. The box plots shows that the median error for GMM++ is less than that of normal GMM. And similar observation is achieved for Kmeans algorithm. So by changing the initialization, the performance of the algorithms improved. This is confirmed by the CHS score as well, the median CHS score for plus plus algorithms(GMM and means) is higher than that of the median CHS score for normal initialization of GMM and kmeans

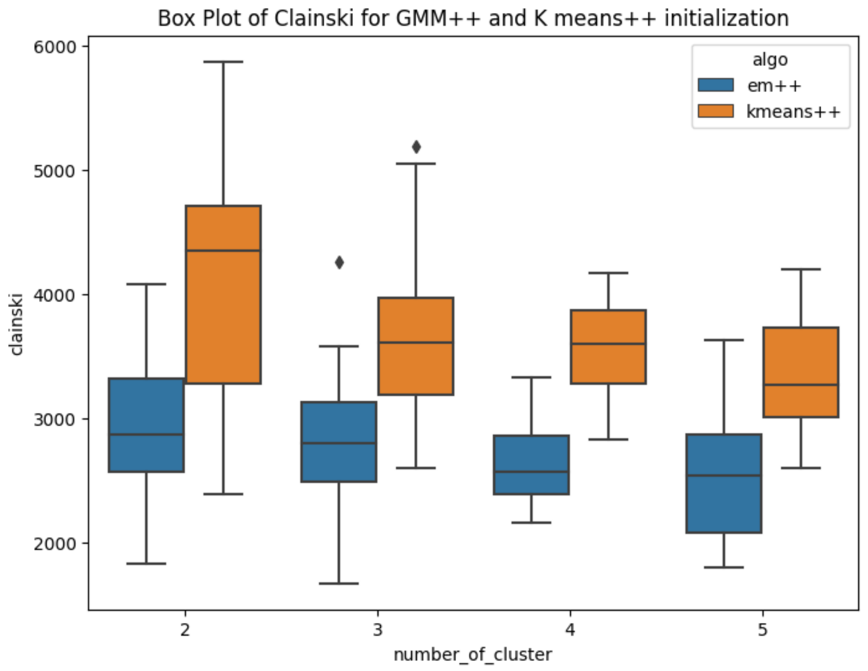
Problem 3 continued on next page...

Plot/s

Place images here with suitable captions.

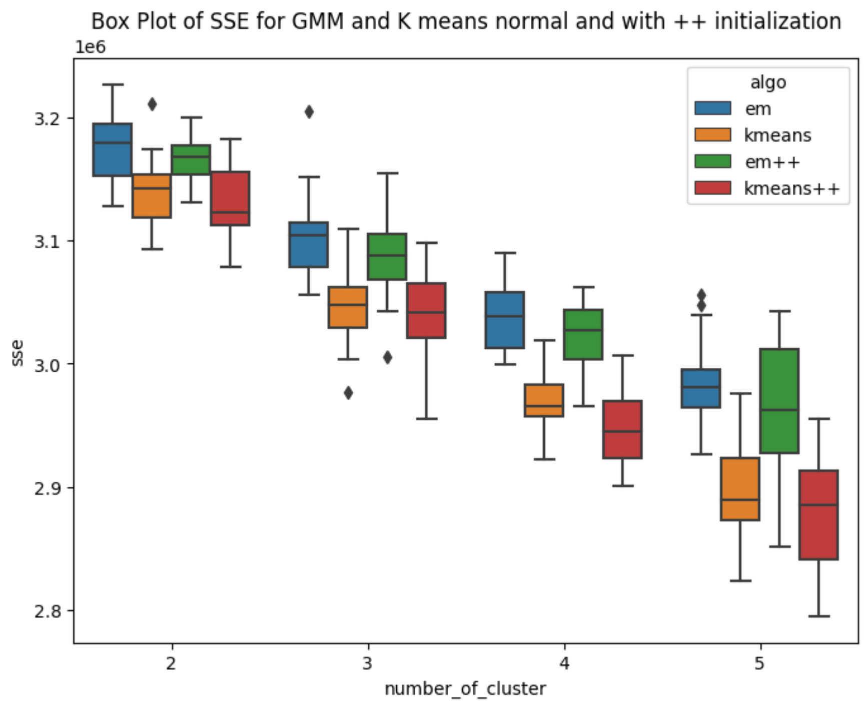


(13)

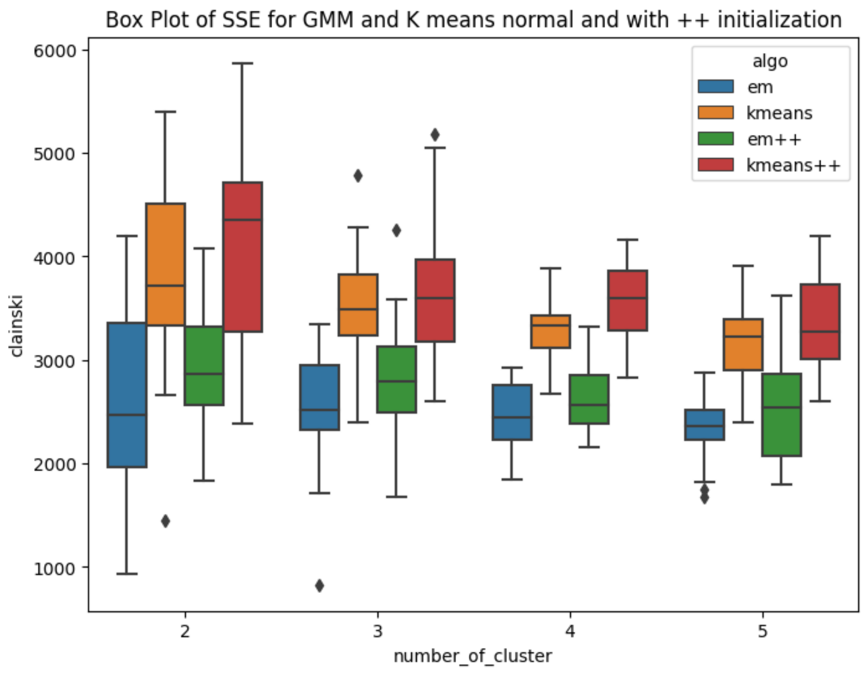


(14)

Problem 3 continued on next page. . .



(15)



(16)

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-hw6-everything.pdf` which you will get after compiling your .tex file.
2. A .zip file with the name `yourname-hw6.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.