`

8.1 Clustering: An Introduction

Clustering is an unsupervised task of grouping data into distinct clusters such that the data-points within a cluster are more similar to each-other than the data-points in other clusters. In process systems, clustering occurs naturally due to multiple reasons. For example, in a power generation plant, production level changes according to the demand leading to significantly different values of plant variables with potentially different inter-variable correlations at different production levels. The multimode nature of data distribution causes problems with traditional ML techniques. To understand this, consider the illustrations in Figure 8.1. In subfigure (a), data indicates 2 distinct modes of operation. From process monitoring perspective, it would make sense to draw separate monitoring boundaries around the two clusters; doing so would clearly identify the red-colored data-point as an outlier or a fault. The Conventional PCA-based monitoring, on the other hand, would fail to identify the outlier. In subfigure (b), the correlation between the variables is different in the two clusters. From soft sensing perspective, it would make sense to build separate models for the two clusters. The Conventional PLS model would give inaccurate results.

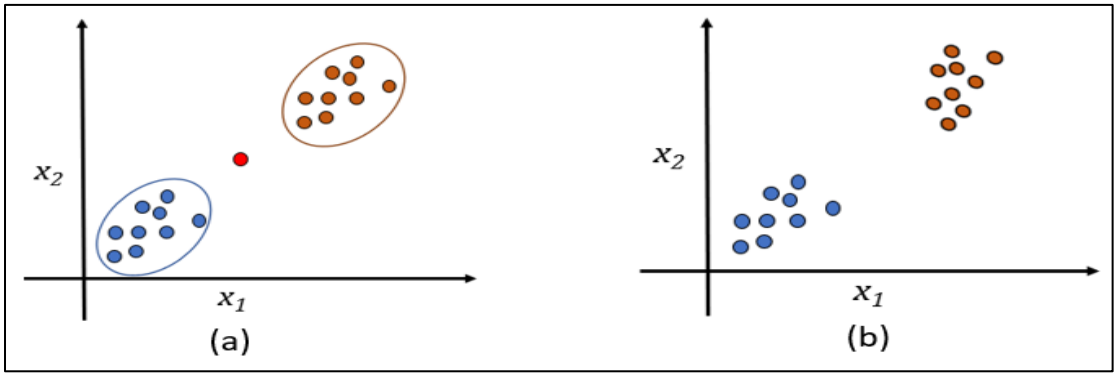

Figure 8.1: Illustrative scenarios for which conventional ML techniques are ill-suited

Once the clusters have been characterized in the training data and cluster-wise models have been built, prediction for a new sample can be obtained by either only considering the cluster-model most suitable for the new sample or combining the predictions from all the models as shown in Figure 8.2. The decision fusion module can also take various forms. For example, for a process monitoring application, a simple fusion strategy could be to consider a new sample as a normal sample if atleast one of the cluster-models predict so. A different strategy could be to combine the abnormality metrics from all the models and make prediction based on this fused metric. Similarly, for soft sensing application, response variable prediction from individual models can be weighted and combined to provide final prediction.

`

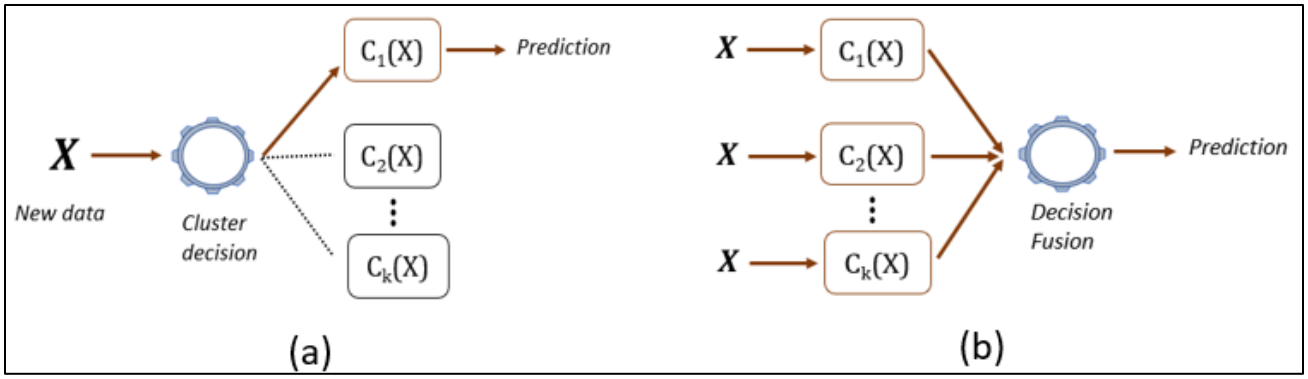

Figure 8.2: ML approaches for multimode processes

There are different clustering algorithms which primarily differ in the way the 'similarity' between the data-points is defined. The popular algorithms can be divided into the following four categories.

- *Centroid-based algorithms*: In these algorithms, the similarity between data-points is quantified by the distance of the data-points from the centroid of the clusters. K-Means, Fuzzy C-Means models belong to this category.

- *Distribution-based algorithms*: Here, similarity between data-points is quantified by computing the probabilities of the data-points belonging to the same underlying distribution. Gaussian mixture models (GMMs) are popular in this category.

- *Density-based algorithms*: Density models find areas of high density of data in the multivariable space and assign data-points to different clusters/regions. DBSCAN and OPTICS are popular examples of density models.

- *Connectivity-based algorithms*: In these models, similarity is quantified by directly measuring the distance between the data-points. Hierarchical clustering models belong to this category.

With so many clustering algorithms at disposal, it is easy to get confused about the right model, especially when dealing with difficult to visualize high-dimensional dataset. In this context, specific domain knowledge, thorough understanding of the advantages, disadvantages, and fundamentals of the algorithms can help you make the right choice.

`

Multimode semiconductor manufacturing operation

In this chapter, we will work with dataset from a semiconductor manufacturing process. The dataset was obtained from multiple batches from an etching process and consists of 19 process variables measured over the course of 108 normal batches and 21 faulty batches. The batch durations range from 95 to 112 seconds. In the rest of the chapter, we will investigate whether the dataset exhibit multimode operations and devise a monitoring strategy to automatically detect the faulty batches. The data is provided in a MATLAB structure array format and so we will use a library to fetch data in Python environment.

```
# import required packages
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans

# fetch data
import scipy.io

matlab_data = scipy.io.loadmat('MACHINE_Data.mat', struct_as_record = False)
Etch_data = matlab_data['LAMDATA']
calibration_dataAll = Etch_data[0,0].calibration # calibration_dataAll[i,0] corresponds to a 2D data from ith batch where columns correspond to different variables

variable_names = Etch_data[0,0].variables

# plot data of a variable for all calibration experiments
plt.figure()
_ = [plt.plot(calibration_dataAll[expt,0][:,6]) for expt in range(calibration_dataAll.size)]
plt.xlabel('Time (s)')
plt.ylabel(variable_names[6])
```

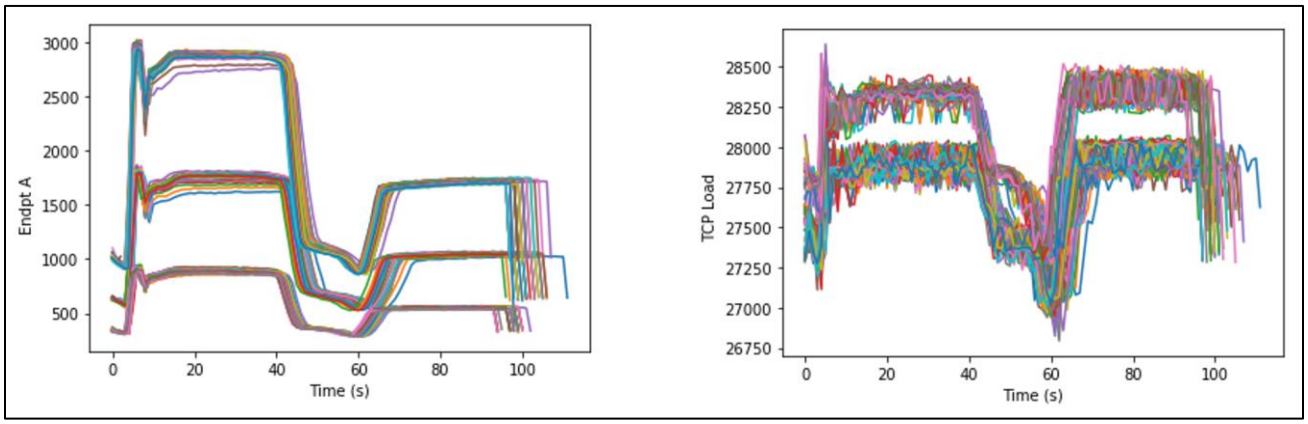

Figure 8.3: Select variable plots for all batches in metal etch dataset. Each colored curve corresponds to a batch.

`

Figure 8.3 does indicate multimode operations with mean and covariance changes. It is however difficult to estimate the number of operation modes by examining high-dimensional dataset directly. A popular practice is to reduce process dimensionality via PCA and then apply clustering to facilitate visualization. Performing PCA serves other purposes as well. We will see later that expectation-maximization (EM) algorithm is employed to estimate cluster parameters in K-Means and GMM models. High dimensionality implies high number of parameters to be estimated which increases possibility of EM converging to locally optimum results and correlated variables cause EM convergence issues. PCA helps to overcome these two problems simultaneously.

We will employ multiway PCA for this batch process dataset. We will follow the approach of He et al.[38] where for each batch 85 sample points are retained to deal with batch length variability, first 5 samples are ignored to eliminate initial fluctuations in sensor measurements, and 3 PCs are retained.

```
# generate unfolded data matrix
n_vars = variable_names.size – 2 # first 2 columns are not process variables
n_samples = 85

unfolded_dataMatrix = np.empty((1, n_vars*n_samples)) # just a placeholder
for expt in range(calibration_dataAll.size):
   calibration_expt = calibration_dataAll[expt,0][5:90,2:]

   if calibration_expt.shape[0] < 85:
      continue

   unfolded_row = np.ravel(calibration_expt, order='F')[np.newaxis,:] # a row for each batch data
   unfolded_dataMatrix = np.vstack((unfolded_dataMatrix, unfolded_row))

unfolded_dataMatrix = unfolded_dataMatrix[1:,:] # remove the empty placeholder

# scale data
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
data_train_normal = scaler.fit_transform(unfolded_dataMatrix)

# PCA
from sklearn.decomposition import PCA
pca = PCA(n_components = 3)
score_train = pca.fit_transform(data_train_normal)
```

[38] He and Wang, Fault detection using the k-nearest neighbor rule for semiconductor manufacturing processes, IEEE Transaction on Semiconductor Manufacturing, 2007

`

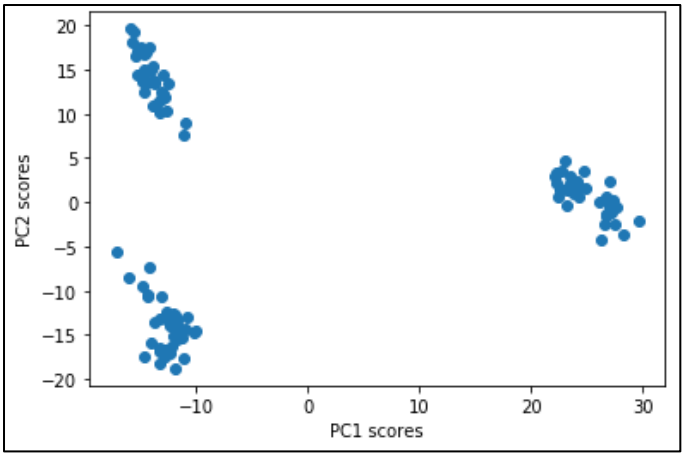

Figure 8.4: Score plot of PC1 and PC2 for calibration batches in metal etch dataset

Figure 8.4 confirms existence of 3 operating modes. While visual inspection of score plots can help to decide the number of clusters, we will, nonetheless, learn ways to estimate this in a more automated way.

8.2 Centroid-based Clustering: K-Means

K-Means is one of the most popular clustering algorithms due to its simple concept, ease of implementation, and computational efficiency. Let *K* denote the number of clusters and $\{\boldsymbol{x}_i\}$, $i = 1, \ldots, N$ be the set of *N* *m*-dimensional points. The cluster assignment of the data points is determined such that the following sum of squared errors, also called cluster inertia, is minimized

$$SSE = \sum_{k=1}^{K} \sum_{\boldsymbol{x}_i \in k^{th} cluster} ||\boldsymbol{x}_i - \boldsymbol{\mu}_k||_2^2 \qquad \textit{eq. 1}$$

Here, $\boldsymbol{\mu}_k$ is the centroid of the k^{th} cluster and $||\boldsymbol{x}_i - \boldsymbol{\mu}_k||_2^2$ denotes the Euclidean distance of $\boldsymbol{x}_i$ from $\boldsymbol{\mu}_k$. To solve Eq 1, k-means adopts the following intuitive iterative procedure.

- Randomly pick K data-points as initial centroids or cluster centers
- Assign each data-point to the closest cluster center
- Recompute the centroids of each cluster using the current cluster assignment of the data-points
- Repeat steps 2 and 3 until convergence

Let us apply k-means to our metal etch dataset.

`

```
# fit k-means model
n_cluster = 3
kmeans = KMeans(n_clusters = n_cluster, random_state = 100).fit(score_train)
cluster_label = kmeans.predict(score_train) # can also use kmeans.labels_

plt.figure()
plt.scatter(score_train[:, 0], score_train[:, 1], c = cluster_label, s = 20, cmap = 'viridis')

cluster_centers = kmeans.cluster_centers_
cluster_plot_labels = ['Cluster ' + str(i+1) for i in range(n_cluster)]
for i in range(n_cluster):
    plt.scatter(cluster_centers[i,0], cluster_centers[i,1], c = 'red', s = 40, marker = '*', alpha = 0.9)
    plt.annotate(cluster_plot_labels[i], (cluster_centers[i,0], cluster_centers[i,1]))
```

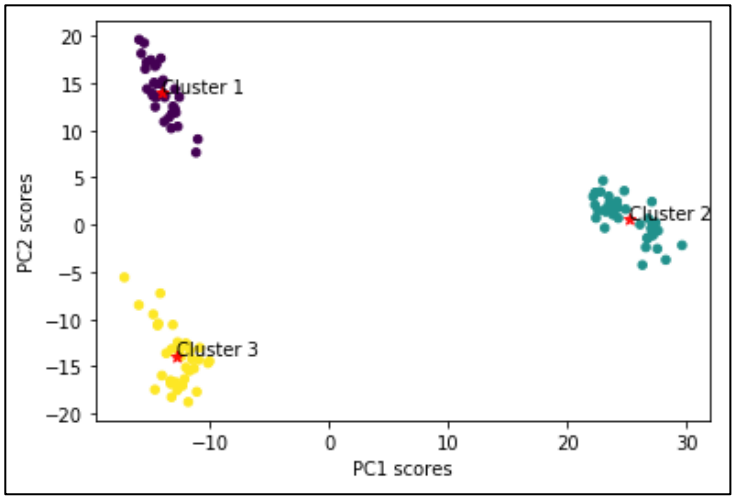

Figure 8.5: Clustering via k-means

As expected, Figure 8.5 shows that k-means does a good job at cluster assignment. K-means clustering results are strongly influenced by initial selection of cluster centers; a bad selection can result in improper clustering. To overcome this, k-means algorithm allows a parameter, n_init (default value is 10), which determines the number of times independent k-means clustering is performed with different initial centroids assignment; the clustering with the lowest SSE is selected as the final model. The strategy for selection of initial centroids can also be changed via init parameter; the default *k-means++* option adopts a smarter (compared to the *random* option) way to speed up convergence by ensuring that the initial centroids are far away from each-other.

`

Determining the number of clusters via elbow method

One of the shortcomings in k-means methodology is the need to specify the number of clusters which is not known a priori for large process datasets. However, we can use the elbow method to overcome this issue. For k-means, the method entails computing the cluster inertia or SSE for different number (*K*) of clusters. Expectedly, when *K* increases, SSE decreases as data-points get closer to their assigned centroids. The value of *K* where only minor improvement in SSE occurs upon increasing *K* can be regarded as an optimal value. Figure 8.6 shows than *K* = 3 is a good choice for metal etch dataset.

```
# determining number of clusters via elbow method
SSEs = []
for n_cluster in range(1, 10):
    kmeans = KMeans(n_clusters = n_cluster, random_state = 100).fit(score_train)
    SSEs.append(kmeans.inertia_)

plt.figure()
plt.plot(range(1,10), SSEs, marker='o')
plt.xlabel('Number of clusters')
plt.ylabel('SSEs')
```

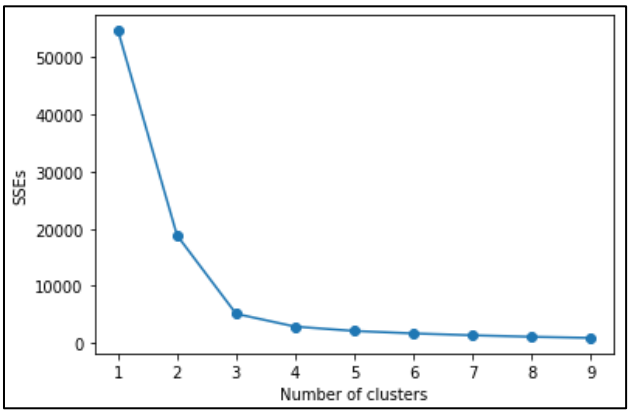

Figure 8.6: Cluster inertias for different number of clusters for metal-etch dataset

Determining the number of clusters or local models should not be taken lightly. We saw in Figure 8.1 that less than optimal number of clusters leads to low soft sensing accuracy and high frequency of missing alarms. On the other hand, using too many clusters increases the number of estimated parameters and consequently, the convergence time. Too many clusters also lead to overfitting, resulting in high frequency of false alarms and low soft sensing accuracy.

`

Silhouette analysis for quantifying clusters quality

For high-dimensional data, it may not always be possible to project most of the data variability onto 2 or 3 PCs and thus it becomes difficult to judge the goodness of clustering or visualize the clusters using 2D or 3D plots. In those situations, silhouette plots can be used to visualize and quantify cluster quality. Note that silhouette analysis is not specific to k-means and can be applied to study any clustering result.

Silhouette coefficient or value of a data-point ranges from -1 to 1 and is a measure of how far the data-point is from data-points in neighboring cluster as compared to data-points in the same cluster. A value of 1 indicates that the data-point is far away from the neighboring cluster and values close to 0 indicate that the data-point is close to the boundary between the two clusters. Negative values indicate wrong cluster assignment.

Figure 8.7 shows the silhouette plot for the cluster shown in Figure 8.5. Each of the colored bands is formed by stacking the silhouette coefficient of all data-points in that cluster and therefore the thickness of the band is an indication of the cluster size. The overall silhouette score is simply the average of silhouette coefficients of all the data-points. As expected, average score is high and cluster 2 shows highest coefficients as it is far away from the other two clusters.

```
# average silhouette score
from sklearn.metrics import silhouette_samples, silhouette_score

silhouette_avgValue = silhouette_score(score_train, cluster_label)
print('Average silhouette score is :', silhouette_avgValue)

>>> Average silhouette score is : 0.7444602567351603

# silhouette plot
from matplotlib import cm

plt.figure()
silhouette_values = silhouette_samples(score_train, cluster_label)
y_lower, y_upper = 0, 0
yticks = []
for i in range(n_cluster):
    cluster_silhouette_vals = silhouette_values[cluster_label == i]
    cluster_silhouette_vals.sort()

    y_upper += len(cluster_silhouette_vals)
    color = cm.nipy_spectral(i / n_cluster)
    plt.barh(range(y_lower, y_upper),cluster_silhouette_vals,height=1.0,edgecolor='none',color=color)
```

`

```
    yticks.append((y_lower + y_upper) / 2)
    y_lower += len(cluster_silhouette_vals)

plt.axvline(silhouette_avgValue, color = "red",  linestyle = "--")
plt.yticks(yticks, np.arange(n_cluster) + 1)
plt.xlabel('Silhouette coefficient values')
plt.ylabel('Cluster')
```

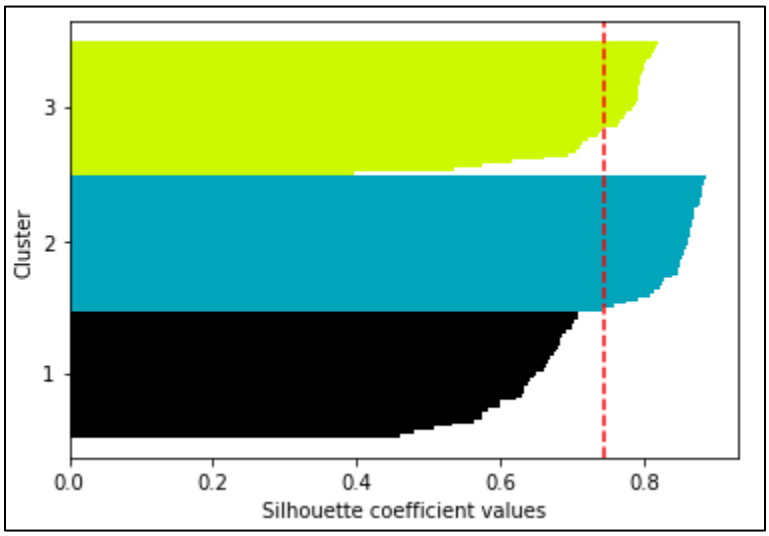

Figure 8.7: Silhouette plot for metal etch data with 3 clusters determined via k-means. Red dashed line denotes the average silhouette value.

For comparison, let's look at a silhouette plot for a sub-optimal clustering in Figure 8.8. Lower sample-wise coefficients and lower overall score clearly indicate worse clustering.

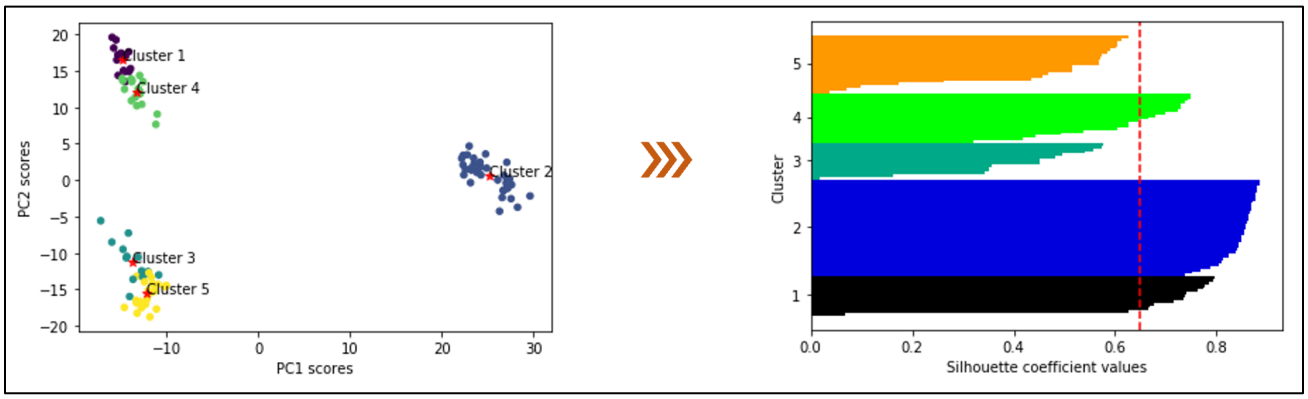

Figure 8.8: Silhouette plot for a sub-optimal clustering example

`

Pros and cons

As alluded to before, ease of application and simple concept are advantages with k-means algorithm. However, a serious shortcoming in this algorithm is the inability to deal with complicated geometries as shown in Figure 8.9. The next set of algorithms that we will study are able to account for such arbitrarily shaped clusters.

```
# generate ellipsoidal shaped data
from sklearn.datasets import make_blobs

n_samples = 1500
X, y = make_blobs(n_samples=n_samples, random_state=100)

rotation_matrix = [[0.60, -0.70], [-0.5, 0.7]]
X_transformed = np.dot(X, rotation_matrix) # elongated blobs

plt.figure()
plt.scatter(X_transformed [:,0], X_transformed [:,1])


# fit k-means model
n_cluster = 3
kmeans = KMeans(n_clusters = n_cluster, random_state = 100).fit(X_transformed)
cluster_label = kmeans.predict(X_transformed)
```

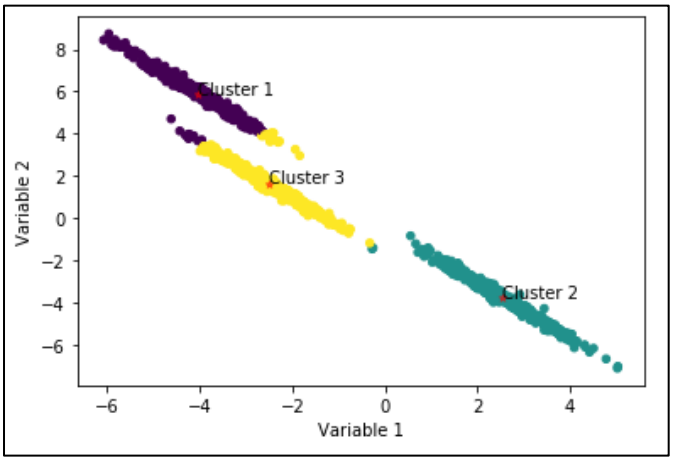

Figure 8.9: k-means clustering failure

`

8.3 Density-based Clustering: DBSCAN

DBSCAN, a Density-based algorithm, can easily handle irregularly shaped data distributions as shown in Figure 8.10 (don't be alarmed by those disconnected dark-brown colored data-points! We will come back to these shortly). DBSCAN works by grouping together data-points that form regions of high data densities. Specifically, each data-point is classified into one of the following 3 categories:

- A **core** point if there are more than a specified number (*minPts*) of data-points within a specified distance (ε)
- A **border** point if less than *minPts* data-points lie within its ε neighborhood, but the data-point itself lies within the ε neighborhood of another core point
- A **noise** point if it isn't classified as either core or border point

After classification into core, border, and noise points, the clusters are defined by the sets of connected core and border data-points. Noise data-points are not assigned to any cluster. The disconnected dark-brown colored data-points that we saw in Figure 8.10 are the noise points. The ability to deal with arbitrarily shaped data, robustness to noise and outliers make DBSCAN well-suited for clustering process data.

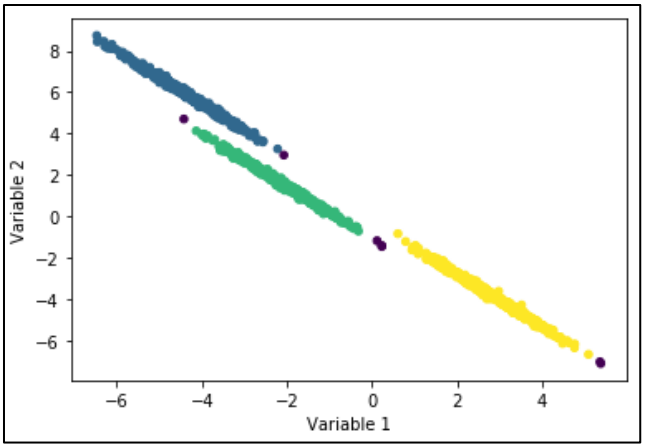

Figure 8.10: DBSCAN succeeds in handling elliptical data distribution

The big shortcoming with the DBSCAN algorithm is the need to specify reasonable values of the two hyperparameters, *minPts* and ε, to ensure optimal clustering. If set improperly, we may end up with either one giant cluster or thousands of small clusters. Let's study this aspect on the metal etch data. Figure 8.11 shows the clustering with properly (*minPts = 3,* ε *= 5*) and improperly (*minPts = 5,* ε *= 3*) set hyperparameters; the improper setting results in several noise data-points and two extra clusters. Trial and error or domain specific knowledge can help to decide the hyperparameter values.

`

```
# fit DBSCAN model to metal-etch data
from sklearn.cluster import DBSCAN

db = DBSCAN(eps = 5, min_samples = 3).fit(score_train)
cluster_label = db.labels_

plt.figure()
plt.scatter(score_train[:, 0], score_train[:, 1], c = cluster_label, s=20, cmap='viridis')
plt.xlabel('PC1 scores')
plt.ylabel('PC2 scores')

print('Cluster labels: ', np.unique(cluster_label))

>>> Cluster labels: [-1 0 1 2 3 4] # noise points are given the label of -1
```

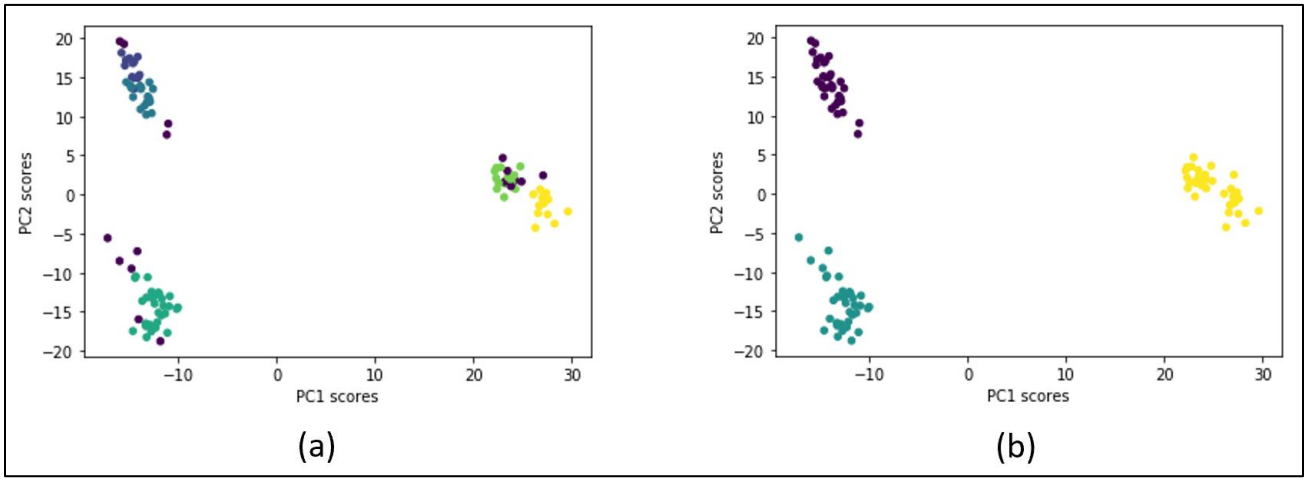

Figure 8.11: Clustering via DBSCAN for metal etch data with (a) improper hyperparameters and (b) proper hyperparameters

Pros and cons

A big advantage of DBSCAN, in addition to those already alluded to, is that the number of clusters does not have to be specified beforehand. The algorithm determines the appropriate number of clusters automatically. A common disadvantage of both k-means and DBSCAN is that the data-points are 'hard-clustered' (each data point assigned to exactly one cluster). For example, consider the data-points categorized as noise-points in Figure 8.10. Although these points are far away from cluster center, they may still correspond to normal operating conditions representing low-likelihood instances. Throwing away these noise points can lead to loss of crucial information and may result in improperly set monitoring thresholds. Additionally, in some cases, hard clustering may not be obvious if a data-point lies equidistant from different clusters. Soft-clustering algorithms can be used to solve these problems as these algorithms provide some measure of association of data-points with different clusters. We will study one such popular algorithm, Gaussian mixture model, in the next section.

`

8.4 Probabilistic Clustering: Gaussian mixtures

While studying PCA and PLS methodologies, we made assumptions about Gaussian distribution of latent variables for control limits determinations. However, as seen for metal etch data, this assumption fails for processes with multiple operating modes. Nonetheless, it may still be appropriate to characterize data from each individual operating mode/cluster through local Gaussian distributions. This is the underlying concept behind Gaussian mixture models (GMMs) and as can be seen in Figure 8.12, it works very well for non-hyperspherical data distributions.

```
# fit GMM model to ellipsoidal data
from sklearn.mixture import GaussianMixture

gmm = GaussianMixture(n_components = 3, random_state = 100)
cluster_label = gmm.fit_predict(X_transformed)

plt.figure()
plt.scatter(X_transformed[:, 0], X_transformed[:, 1], c = cluster_label, s=20, cmap='viridis')

cluster_centers = gmm.means_ # cluster centers
cluster_plot_labels = ['Cluster ' + str(i+1) for i in range(gmm.n_components)]
for i in range(gmm.n_components):
    plt.scatter(cluster_centers[i, 0], cluster_centers[i, 1], c='red', s=20, marker = '*', alpha=0.5)
    plt.annotate(cluster_plot_labels[i], (cluster_centers[i,0], cluster_centers[i,1]))
```

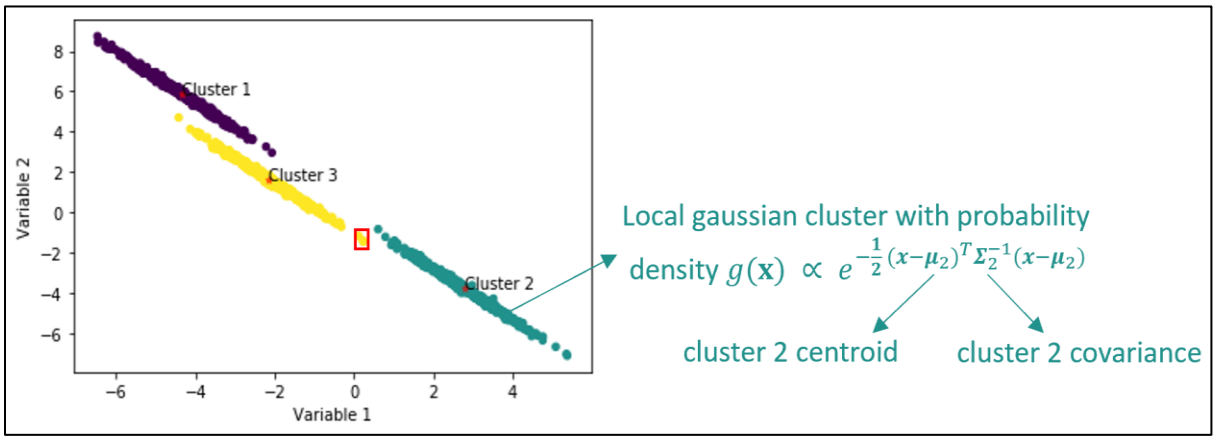

Figure 8.12: GMM based clustering of ellipsoidal data distribution

Another big advantage with GMMs is that we can compute the (posterior) probability of a data-point belonging to any cluster. This cluster membership measure is provided by predict_proba method. Hard clustering is performed by assigning the data-point to the

`

cluster with highest probability. Let's compute the probabilities for a data-point that lies between clusters 3 and 2 (encircled in Figure 8.12).

```
# membership probabilities
probs = gmm.predict_proba(X_transformed[1069, np.newaxis]) # requires 2D array
print('Posterior probabilities of clusters 1, 2, 3 for the data-point: ', probs[-1,:])

>>> Posterior probabilities of clusters 1, 2, 3 for the data-point:  [3.36e-54 1.09e-15 1]
```

GMM thinks that the data-point belongs to cluster 3 with 100% probability! It may seem surprising given that the point seems to lie equidistant (in terms of Euclidean distance) to clusters 3 and 2. We will study in the next subsection how these probabilities were obtained.

Mathematical background

Let $\boldsymbol{x} \in \mathbb{R}^m$ be a m-dimensional sample obtained from a multimode process with *K* operating modes. In GMM, the overall probability density is formulated as a combination of local Gaussian densities. Let C_i denote the i^{th} local Gaussian cluster with parameters $\boldsymbol{\theta}_i = \{\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i\}$ (mean vector and covariance matrix) and density

$$\mathrm{g}(\mathbf{x}|\boldsymbol{\theta}_i) = \frac{1}{(2\pi)^{m/2}\,|\boldsymbol{\Sigma}_i|^{1/2}} exp\left[-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_i)^T \boldsymbol{\Sigma}_i^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_i)\right] \quad \text{eq. 2}$$

The overall density at any spatial location is given by

$$p(\boldsymbol{x}|\boldsymbol{\theta}) = \sum_{i=1}^{K} \omega_i g(\boldsymbol{x}|\boldsymbol{\theta}_i) \quad \textit{eq. 3}$$

where, ω_i represents the prior probability that a new sample comes from the i^{th} Gaussian component and $\boldsymbol{\theta} = \{\theta_1, \ldots, \theta_k\}$. The GMM model is constructed by estimating the parameters $\boldsymbol{\theta}_i$, $\boldsymbol{\omega}_i$ for all the clusters using the training samples $\boldsymbol{X} \in \mathbb{R}^{N\,X\,m}$. The parameters are estimated by optimizing the log-likelihood of the training dataset given as below

$$\sum_{j=1}^{N} log\left(\sum_{i=1}^{K} \omega_i \mathrm{g}(\boldsymbol{x}_j|\boldsymbol{\theta}_i)\right) \quad \text{eq. 4}$$

To optimize the likelihood, expectation-maximization (EM) algorithm is commonly employed. Assuming an initial estimate of the parameters are given, EM algorithm involves iterating between 2 steps:

- E-step (s^{th} iteration): (Re-)compute membership association of samples to local clusters

$$P^{(s)}(C_i|\boldsymbol{x}_j) = \frac{\omega_i^{(s)} g(\boldsymbol{x}_j \mid \boldsymbol{\mu}_i^{(s)}, \boldsymbol{\Sigma}_i^{(s)})}{\sum_{k=1}^{K} \omega_k^{(s)} g(\boldsymbol{x}_j \mid \boldsymbol{\mu}_k^{(s)}, \boldsymbol{\Sigma}_k^{(s)})} \quad \text{eq. 5}$$

`

$P^{(s)}(C_i|\boldsymbol{x}_j)$ denotes the posterior probability that the *j*th sample comes from the *i*th Gaussian component.

- M-step ((*s+1)*th iteration): Update GMM parameters

$$\boldsymbol{\mu_i^{(s+1)}} = \frac{\sum_{j=1}^{N} P^{(s)}(C_i|\boldsymbol{x_j})\boldsymbol{x_j}}{\sum_{j=1}^{N} P^{(s)}(C_i|\boldsymbol{x_j})}$$

$$\boldsymbol{\Sigma_i^{(s+1)}} = \frac{\sum_{j=1}^{N} P^{(s)}(C_i|\boldsymbol{x_j})(\boldsymbol{x_j}-\boldsymbol{\mu_i^{(s+1)}})(\boldsymbol{x_j}-\boldsymbol{\mu_i^{(s+1)}})^T}{\sum_{j=1}^{N} P^{(s)}(C_i|\boldsymbol{x_j})}$$

$$\omega_i^{(s+1)} = \frac{\sum_{j=1}^{N} P^{(s)}(C_i|\boldsymbol{x_j})}{N}$$

Update centroid and covariance of each cluster using recomputed memberships from E-step.

The iteration continues until some convergence criterion on log-likelihood objective is met. Did you notice the conceptual similarity with the k-means algorithm for finding model parameters? Previously, we computed posterior probabilities for data point 1069 using predict_prob method. Let us now use eq. 5 to see if we get the same numbers.

```
# posterior probability calculation
x = X_transformed[1069,np.newaxis]

import scipy.stats
g1 = scipy.stats.multivariate_normal(gmm.means_[0,:], gmm.covariances_[0,:]).pdf(x)
g2 = scipy.stats.multivariate_normal(gmm.means_[1,:], gmm.covariances_[1,:]).pdf(x)
g3 = scipy.stats.multivariate_normal(gmm.means_[2,:], gmm.covariances_[2,:]).pdf(x)
print('Local component densities: ', g1, g2, g3)

>>> Local component densities:  8.33e-56 2.72e-17 0.025

den = gmm.weights_[0]*g1 + gmm.weights_[1]*g2 + gmm.weights_[2]*g3
posterior_prob_cluster1 = gmm.weights_[0]*g1/den
posterior_prob_cluster2 = gmm.weights_[1]*g2/den
posterior_prob_cluster3 = gmm.weights_[2]*g3/den
print('Posterior probabilities: ', posterior_prob_cluster1, posterior_prob_cluster2, posterior_prob_cluster3)

>>> Posterior probabilities:  3.36e-54 1.09e-15 1
```

As expected, we obtain the same posterior probabilities. Although component-wise densities are low, component 3 has (relatively) much higher density value at spatial coordinates of point 1069 and hence, resulting in highest membership association! Hopefully, this quick calculation helped you gain more insights into the workings of GMM models.

`

Determining the number of clusters

One of the limitations with EM algorithm for GMM modeling is that the number of Gaussian components is assumed to be known beforehand. This may not always be true. To overcome this, we can use a method similar to elbow method that we used for k-means. While in k-means, we plotted SSEs for different number of clusters, in GMM we will utilize the Bayesian Information Criterion (BIC) metric (introduced in Chapter 3). Unlike SSEs, BICs increase after a while and optimal Gaussian components is the value that minimizes BIC.

```
# finding # of components via BIC method
BICs = []
lowestBIC = np.inf
for n_cluster in range(1, 10):
    gmm = GaussianMixture(n_components = n_cluster, random_state = 100)
    gmm.fit(X_transformed)
    BIC = gmm.bic(X_transformed)
    BICs.append(BIC)

    if BIC < lowestBIC:
        optimal_n_cluster = n_cluster
        lowestBIC = BIC

plt.figure(), plt.plot(range(1,10), BICs, marker='o')
plt.scatter(optimal_n_cluster, lowestBIC, c='red', marker='*', s=1000)
```

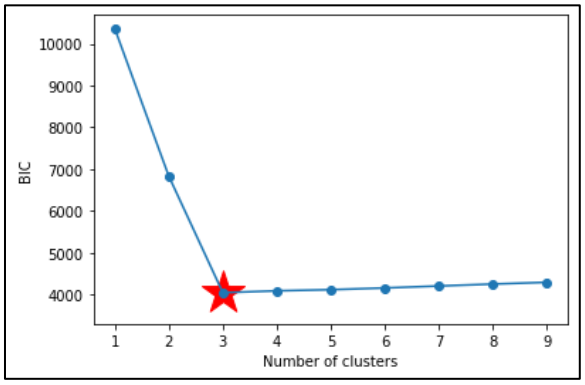

Figure 8.13: BIC plot for determining number of GMM components

There is another method, called F-J algorithm[39], which can be used to find the optimal number of GMM components[40] and model parameters in an integrated manner. Also, the number of components does not need to be specified beforehand in F-J method. Internally, the method initializes with a large number of components and adaptively adjusts this number by

[39] Figueiredo & Jain, Unsupervised learning of finite mixture models, *IEEE Trans Pattern Anal Mach Intell*, 2002

[40] Yu & Qin, Multimode process monitoring with Bayesian inference-based finite gaussian mixture models, AIChE Journal, 2008

`

eliminating Gaussian components with insignificant weights. F-J method also utilizes EM algorithm for parameter estimation, but with a slightly different weight update mechanism in the M-step. The reader is encouraged to see the cited references for more details. A downside of F-J method could be high computational time.

```
# finding # of components via FJ algorithm
from gmm_mml import GmmMml
gmmFJ = GmmMml(plots=False)
gmmFJ.fit(X_transformed)
cluster_label = gmmFJ.predict(X_transformed)

plt.figure()
plt.scatter(X_transformed[:, 0], X_transformed[:, 1], c = cluster_label, s=20, cmap='viridis')
```

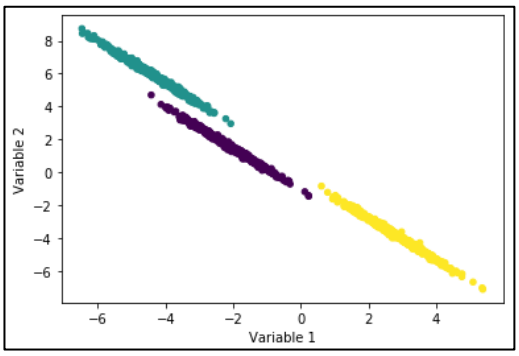

Figure 8.14: GMM based clustering of ellipsoidal data distribution via F-J method

Figure 8.15 shows the clustering for metal etch data via GMM method. BIC method correctly identifies the optimal number of components. Note that F-J method results in 4 components for this dataset.

```
# fit GMM model to metal-etch data
gmm = GaussianMixture(n_components = optimal_n_cluster, random_state = 100)
cluster_label = gmm.fit_predict(score_train)
plt.scatter(score_train[:, 0], score_train[:, 1], c = cluster_label, s=20, cmap='viridis')
```

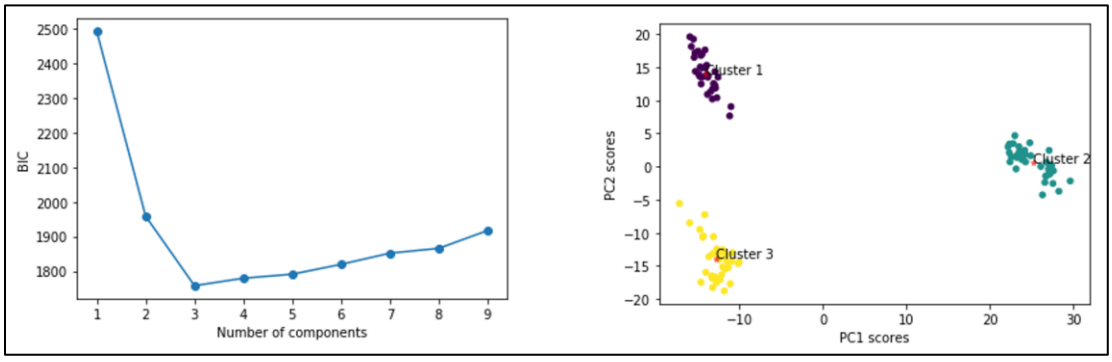

Figure 8.15: BIC plot and GMM clustering of metal etch data

`

8.5 Multimode Process Monitoring via GMM for Semiconductor Manufacturing Process

Due to probabilistic formulation, GMM is widely applied for monitoring process systems. In this section, we will study one such application for the metal etch process. Figure 8.16 shows the metal etch calibration and faulty batches in the PCA score space. It is apparent that the faulty batches tend to lie away from the calibration clusters. Our objective is to develop a GMM-based monitoring tool that can automatically detect these faulty batches.

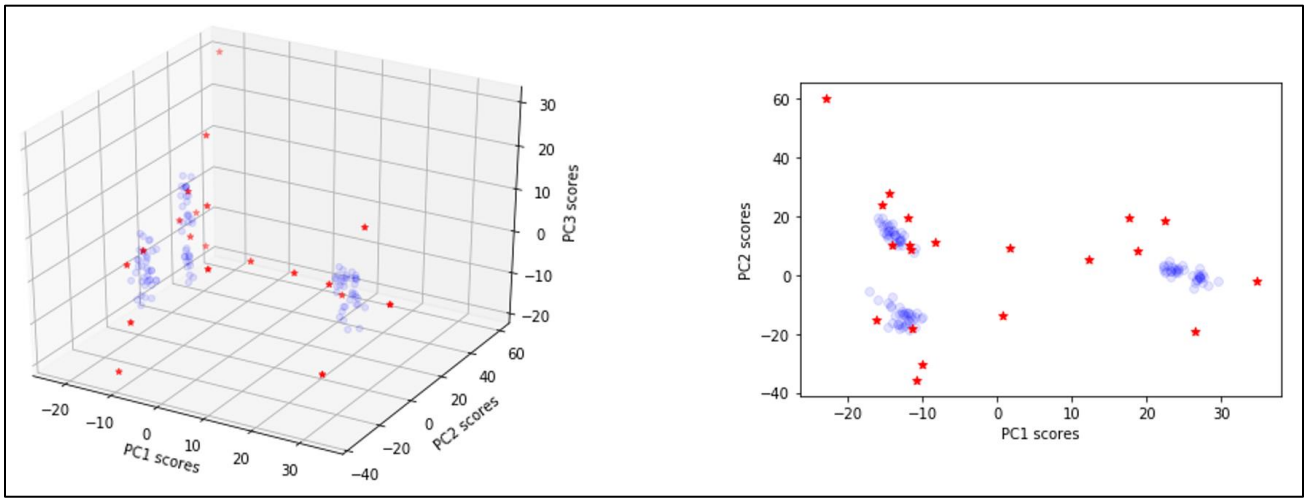

Figure 8.16: Calibration (in blue) and faulty (in red) batches in PCA score space

Process monitoring/Fault detection index

A straightforward approach to detect a faulty batch would be to build separate ellipsoidal boundary or T^2 control chart for each cluster/mode and check if the T^2 metric for the faulty batch falls outside the control limits for all the clusters. However, monitoring many control charts can become tedious when the number of Gaussian components is high. For convenience, the local metrics/charts can be combined to generate a single global metric/chart. We will follow the approach of Xie & Shi[41] (an alternative approach is provided by Yu & Qin[39]). For a test data point, $\boldsymbol{x}_t$, local Mahalanobis distance is computed from each cluster

$$D_{local}^{(k)}(\boldsymbol{x}_t) = (\boldsymbol{x}_t - \boldsymbol{\mu}_k)^T \boldsymbol{\Sigma}_k^{-1} (\boldsymbol{x}_t - \boldsymbol{\mu}_k) \qquad \text{eq. 6}$$

[41] Xie & Shi, Dynamic multimode process modeling and monitoring using adaptive Gaussian mixture models, Industrial & Engineering Chemistry Research, 2012

`

The global metric is then computed using the posterior probabilities of the test sample to each Gaussian component

$$D_{global}(\boldsymbol{x}_t) = \sum_{k=1}^{K} P(C_k|\boldsymbol{x}_t)\, D_{local}^{(k)}(\boldsymbol{x}_t) \quad \text{eq. 7}$$

The control limit for D_{global} can be obtained using an *F*-distribution

$$D_{global,CL} = \frac{r(N^2-1)}{N(N-r)}\, F_{r,N-r}(\alpha) \quad \text{eq. 8}$$

$F_{r,N-r}(\alpha)$ is the *(1-α)* percentile of the *F*-distribution with *r* and *n-r* degrees of freedom, *r* is variable dimension (we performed GMM in PCA score space with 3 latent variables, therefore, *r* = 3). Test sample is considered abnormal if $D_{global} > D_{global,\,CL}$.

```
# global Mahalanobis distance metric
Dglobal_train = np.zeros((score_train.shape[0],))

for i in range(score_train.shape[0]):
    x = score_train[i,:,np.newaxis]
    probs = gmm.predict_proba(x.T)

    for component in range(3):
        Dlocal = np.dot(np.dot((x-gmm.means_[component,:,np.newaxis]).T,
                               np.linalg.inv(gmm.covariances_[component,:])),
                        (x-gmm.means_[component,:,np.newaxis]))

        Dglobal_train[i] =  Dglobal_train[i] + probs[0,component]*Dlocal

# Dglobal control limit
N = score_train.shape[0]
r = 3
alpha = 0.05 # 95% control limit
Dglobal_CL = r*(N**2-1)*scipy.stats.f.ppf(1-alpha, r, N-r)/(N*(N-r))

# Dglobal control chart
plt.figure()
plt.plot(Dglobal_train)
plt.plot([1, len(Dglobal_train)], [Dglobal_CL, Dglobal_CL], color='red')
plt.xlabel('Sample #'), plt.ylabel('D_global for training data')
```

Figure 8.17 shows the control chart for the training data.

`

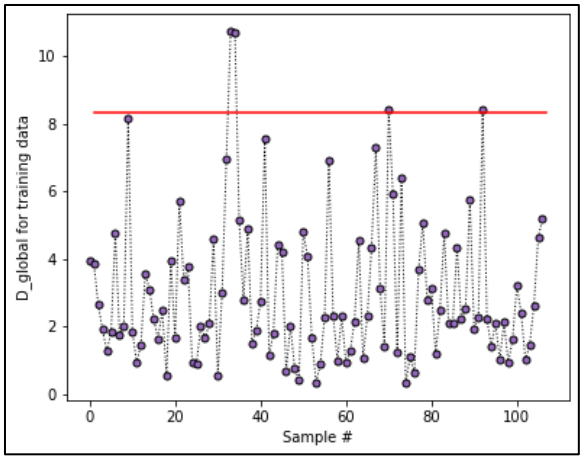

Figure 8.17: Global monitoring chart for metal etch calibration data

Fault detection for test data

Figure 8.18 shows that the global distance metric seems to be a suitable metric in detecting the faulty batches. The monitoring metric violates the threshold for 18 out of 20 batches. This good monitoring performance can be attributed to the proper multimode modeling of the data.

```
# fetch test data and unfold
test_dataAll = Etch_data[0,0].test

unfolded_TestdataMatrix = np.empty((1,n_vars*n_samples))
for expt in range(test_dataAll.size):
   test_expt = test_dataAll[expt,0][5:90,2:]

   if test_expt.shape[0] < 85:
      continue

   unfolded_row = np.ravel(test_expt, order='F')[np.newaxis,:]
   unfolded_TestdataMatrix = np.vstack((unfolded_TestdataMatrix, unfolded_row))

unfolded_TestdataMatrix = unfolded_TestdataMatrix[1:,:]

# PCA on faulty data
data_test_normal = scaler.transform(unfolded_TestdataMatrix)
score_test = pca.transform(data_test_normal)

# compute Dglobal_test
Dglobal_test = np.zeros((score_test.shape[0],))

for i in range(score_test.shape[0]):
   x = score_test[i,:,np.newaxis]
```

`

```
    probs = gmm.predict_proba(x.T)

for component in range(3):
        Dlocal = np.dot(np.dot((x-gmm.means_[component,:,np.newaxis]).T,
np.linalg.inv(gmm.covariances_[component,:])), (x-gmm.means_[component,:,np.newaxis]))

        Dglobal_test[i] =  Dglobal_test[i] + probs[0,component]*Dlocal

print('Number of faults identified: ', np.sum(Dglobal_test > Dglobal_CL), ' out of ',
len(Dglobal_test))

>>> Number of faults identified:  18  out of  20
```

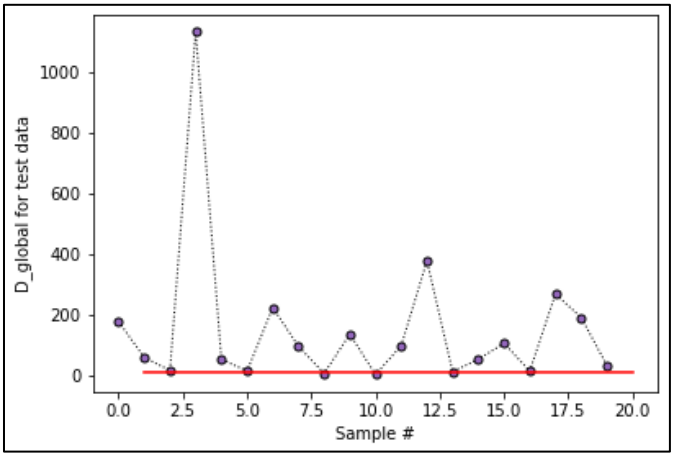

Figure 8.18: Global monitoring chart for test data

Summary

With this chapter we have conquered another milestone in our journey. We started by understanding the need for multimode modeling of process systems and then studied several popular clustering/mixture modeling algorithms. The emphasis on the pros and cons of different methods was a deliberate choice to enable you to make educated selection of algorithms for your problems. You are going to encounter multimode processes frequently in your career and the tools studied in this chapter will help you analyze these systems properly.