`

6.1 ICA: An Introduction

Independent Component Analysis (ICA) is a multivariate technique for transforming measured variables into statistically independent latent variables in a lower-dimensional space. Statistical independence is a stricter condition than uncorrelatedness and in some situations, working with independent components (ICs) can give better results than working with uncorrelated PCs from PCA. While ICA and PCA are related (in the sense that latent variables are linear projections of measured variables), they differ in the way the latent variables are extracted. Figure 6.1 highlights the difference between them using a simple illustration where two independent signals are linearly combined to generate correlated signals and then PCA/ICA are used to extract latent signals. It is apparent that simply decorrelating the signals via PCA did not recover the original signals. On the other hand, ICA reconstructs the original signals accurately. If you observe closely, you will find that ICA latent signals (u1 and u2) do differ from s1 and s2 signals in terms of sign and magnitude; we will soon learn why this happens and why this is not a cause of worry.

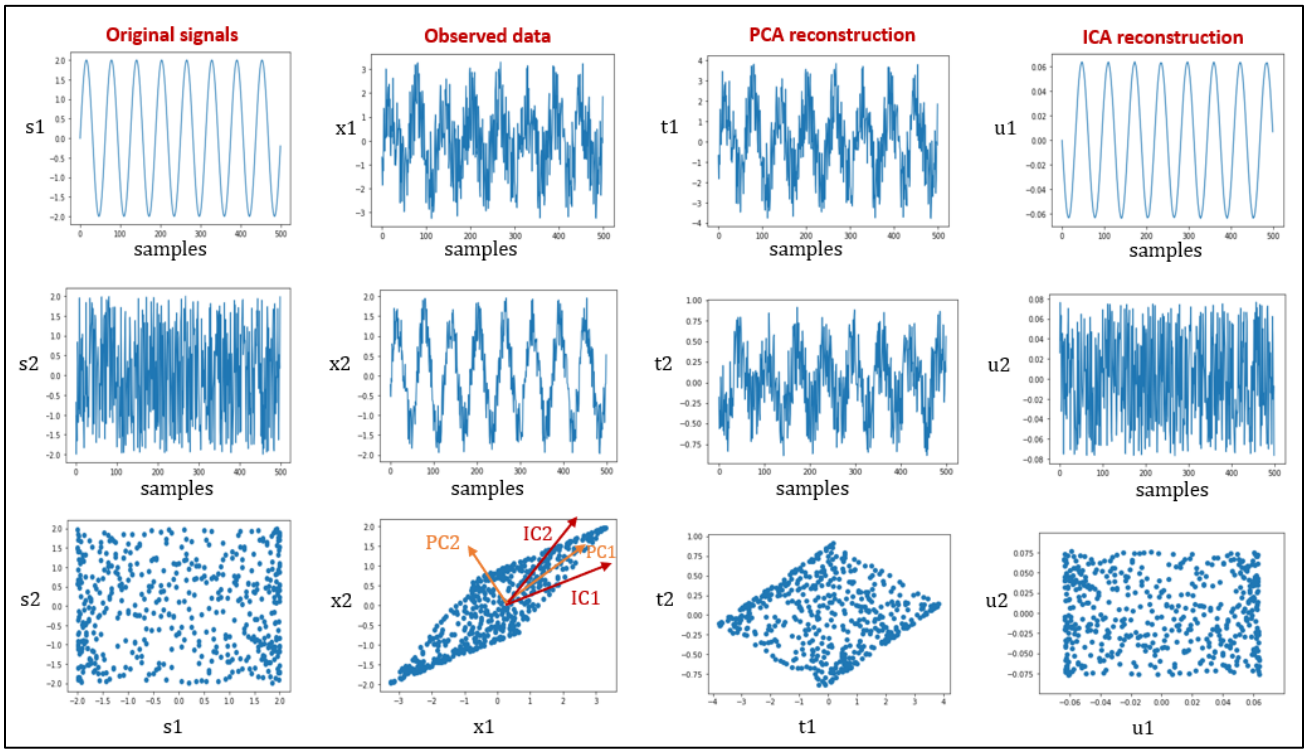

Figure 6.1: Simple illustration of ICA vs PCA. The arrows in the x1 vs x2 plot show the direction vectors of corresponding components. Note that the signals t1 and t2 are not independent as value of one variable influences the range of values of the other variable.

ICA uses higher-order statistics for latent variable extractions, instead of only second order statistics (mean, variance/covariance) as done by PCA. Therefore, for non-Gaussian

`

(multivariate Gaussian distributions are completely defined by second-order statistics) industrial datasets, ICs can provide more useful information than PCs, resulting in potentially better modeling performance. In the above illustration we saw how ICA provided more meaningful representation by aligning the IC direction vectors along the edges of the data-cluster, while PC vectors were compelled, by design, to point in the direction of maximal variance. Figure 6.2 summarizes the major differences between ICA and PCA. By the end of this chapter, you will understand these subtle differences and their implications more clearly.

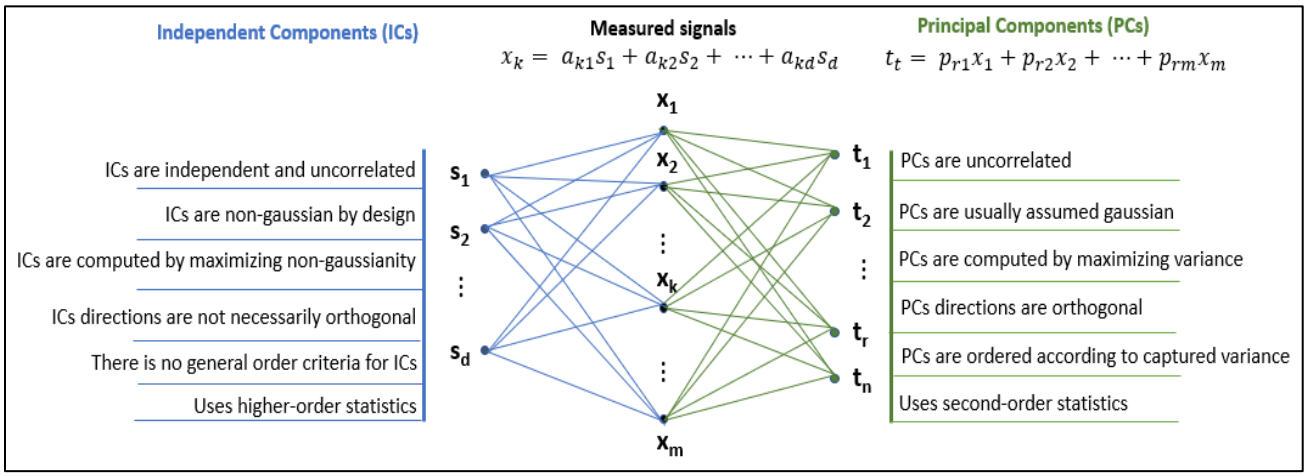

Figure 6.2: Differences between ICA and PCA

Independence vs Uncorrelatedness

Before jumping into the mathematics behind ICA, let us take a few seconds to ensure that we understand the concepts behind independence and uncorrelatedness. Two random variables y1 and y2, are said to be independent if the value of one signal does not impact the value of the other signal. Mathematically, this condition is stated as

$$p(y1, y2) = p(y1)p(y2)$$

Where $p(y1)$ is the probability density function of y1 alone, and $p(y1, y2)$ is the joint probability density function. The variables y1 and y2 are said to be uncorrelated if their covariance is zero

$$C(y1, y2) = E\{(y1 - E(y1))(y2 - E(y2))\} = E(y1 * y2) - E(y1)E(y2) = 0$$

Where $E(.)$ denotes mathematical expectation. Using the independence condition, it can be easily shown that if the variables are independent, they are also uncorrelated but not vice versa. Therefore, uncorrelatedness is a weaker form of independence.

`

Mathematical background

Consider data matrix $\boldsymbol{X} \in \mathbb{R}^{m \times \mathrm{N}}$ consisting of *N* observations of *m* input variables where each column represents a data-point in the original measurement space. Note that in contrast to PCA, transposed form of data matrix is employed here. In ICA, it is assumed that measured variables are a linear combination of *d* (≤ *m*) independent components $s_1, s_2, \ldots, s_d$.

$$\boldsymbol{X} = \boldsymbol{AS} \qquad \text{eq. 1}$$

where $\boldsymbol{S} \in \mathbb{R}^{d \times \mathrm{N}}$ and $\boldsymbol{A} \in \mathbb{R}^{m \times \mathrm{d}}$ is called the mixing matrix. The objective of ICA is to estimate the unknown matrices ***A*** and ***S*** from the measured data ***X***. This is accomplished by finding a demixing matrix, ***W***, such that the ICs or the rows of estimated matrix ($\widehat{\boldsymbol{S}}$) become as independent as possible.

$$\widehat{\boldsymbol{S}} = \boldsymbol{WX} \qquad \text{eq. 2}$$

Before estimating ***W***, the initial step in ICA involves removing correlations between the variables in the data matrix ***X***. The step, called as whitening or sphering, is accomplished via PCA.

$$\boldsymbol{Z} = \boldsymbol{QX} \qquad \text{eq. 3}$$

where $\boldsymbol{Q} \in \mathbb{R}^{d \times \mathrm{d}}$ is called whitening matrix. Whitening makes the rows of ***Z*** uncorrelated. To see how it helps, let ***B*** = ***QA*** and consider the following relationships,

$$\boldsymbol{Z} = \boldsymbol{QX} = \boldsymbol{QAS} = \boldsymbol{BS} \qquad \text{eq. 4}$$

Therefore, whitening converts the problem of finding matrix A into that of finding matrix B. The advantage lies in the fact that B is an orthogonal matrix - can be shown considering that whitened variables are uncorrelated and ICs are independent - and hence fewer parameters need to be estimated. Using orthogonality property, the following relationship results,

$$\begin{aligned} \boldsymbol{S} &= \boldsymbol{B^T Z} = \boldsymbol{B^T QX} \\ \Rightarrow \boldsymbol{W} &= \boldsymbol{B^T Q} \end{aligned} \qquad \text{eq. 5}$$

Therefore, once ***B*** is found, ***W*** can be generated. Matrix ***B*** is estimated by maximizing the non-Gaussianity of the estimated IC values, i.e., each row of ***S*** ($\boldsymbol{s_i}$) should be maximally away from a Gaussian distribution. Using Central Limit Theorem, it can be shown that maximizing non-Gaussianity results in independent components[27] that we need.

The above procedure summarizes the steps involved in ICA. Note that the sets {***A***/n, ***S***/n} and {***A***, ***S***} result in the same measured data matrix ***X*** for any non-zero scalar *n*, and therefore the sign and magnitude of the original ICs cannot be uniquely estimated. This however does not

[27] www.sci.utah.edu/~shireen/pdfs/tutorials/Elhabian_ICA09.pdf is an excellent quick read for more details on this.

`

affect usage of ICA for process monitoring because the estimated ICs for both training and test data get scaled by the same scalar implicitly. FastICA (a popular algorithm for ICA) computes ICs such that the L_2 norm of each IC score is 1. This is the reason why the reconstructed IC signals in Figure 6.1 seem to be scaled versions of original IC signals. We will use this fact later, so do not forget it!

Complex chemical process: Tennessee Eastman Process (TEP)

In this chapter, we will use dataset from a large-scale industrial chemical plant to demonstrate the superior performance of ICA and FDA over PCA. Details on this TEP dataset can be found in Appendix. The process consists of several unit operations, and has 22 continuous process measurements, 19 composition measurements, and 11 manipulated variables. The dataset contains training and test data from normal operation period and 21 faulty periods with distinct fault causes. For each fault class, training datasets contains 480 samples and test datasets contains 960 samples. For the faulty data, faulty operation starts from sample 160 onwards.

Let's quickly see the process impact of one of the fault conditions (fault 10) which include disturbances in one of the feed's temperature. The impact of this fault can be seen in abnormal stripper temperature profile (Figure 6.3a). The plot in PC space (Figure 6.3b) shows more clearly how the faulty operation data are different from the normal operation data. We will use ICA and FDA for detecting and classifying these faults automatically.

```
# import required packages
import numpy as np
import matplotlib.pyplot as plt

# fetch TE data
TEdata_noFault_train = np.loadtxt('d00.dat').T
TEdata_Fault_train = np.loadtxt('d10.dat')

# quick visualization
plt.figure()
plt.plot(TEdata_noFault_train[:,17])
plt.xlabel('sample #'), plt.ylabel('Striper Tempearture'), plt.title('Normal operation')

plt.figure()
plt.plot(TEdata_Fault_train[:,17])
plt.xlabel('sample #'), plt.ylabel('Striper Tempearture'), plt.title('Faulty operation')
```

`

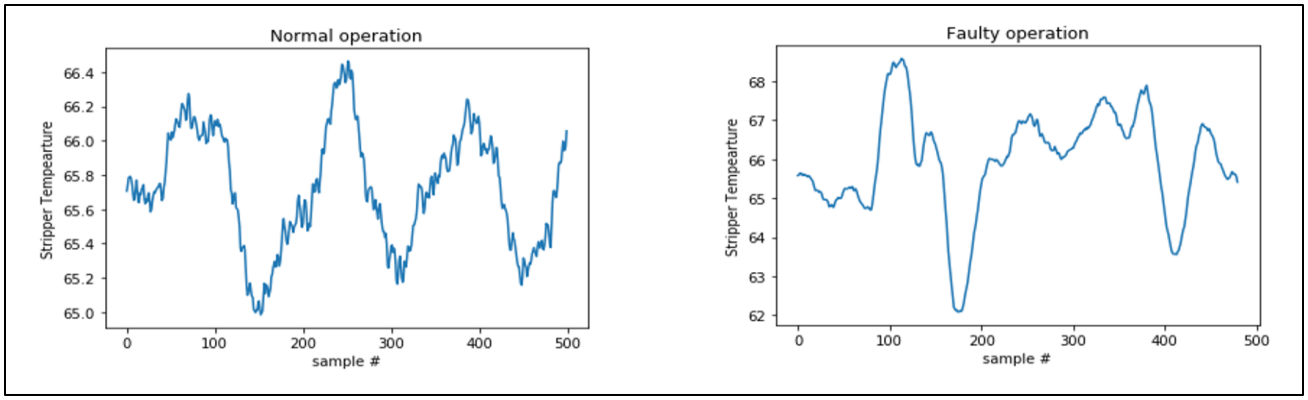

Figure 6.3 (a): Normal vs faulty process profile in Tennessee Eastman dataset

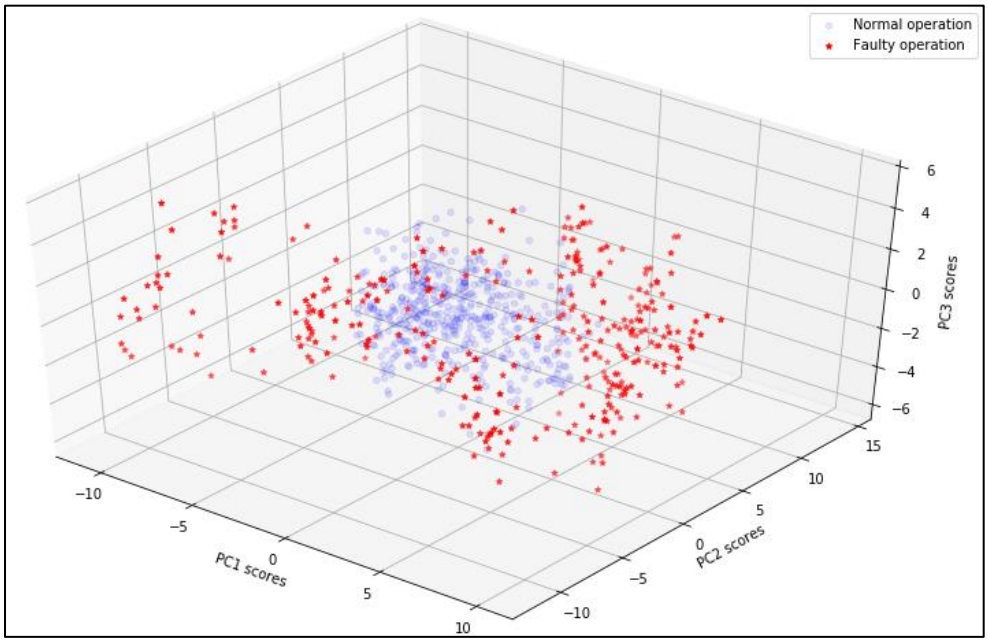

Figure 6.3 (b): Normal vs faulty (Fault 10) TE process data in PC space

Deciding number of ICs

The number of ICs returned by FastICA equals the dimension of the measured space. Previously, we saw in PCA that not all extracted latent variables are important; only a few dominant/important components contain majority of the information while the rest contain noise or trivial details. Moreover, the extracted PCs were ordered according to their variance. However, there is no standard criterion for quantifying the importance of ICs and selecting the optimal number of ICs automatically. One popular approach (described below) for quantifying component importances was suggested by Lee et. al[28].

[28] Lee et al., Statistical process monitoring with independent component analysis, Journal of Process Control, 2004

`

Equation 2 shows that the i^{th} row ($\boldsymbol{w}_i$) of demixing matrix $\boldsymbol{W}$ corresponds to the i^{th} IC. Therefore Lee et al. suggested using the Euclidean norm (L_2) of $\boldsymbol{w}_i$ to quantify the importance of i^{th} IC and subsequently order the ICs in decreasing order of importance. Using this rationale, Figure 6.4 shows that not all ICs are equally important as the L_2 norm of several ICs are much smaller than the rest.

```
# fetch TE data and select variables (discarding composition measurements)
TEdata_noFault_train = np.loadtxt('d00.dat').T

xmeas = TEdata_noFault_train[:,0:22] # 22 continuous process variables
xmv = TEdata_noFault_train[:,41:52] # 11 manipulated variables
data_noFault_train = np.hstack((xmeas, xmv))

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

# fit ICA model
from sklearn.decomposition import FastICA
ica = FastICA(max_iter=1000, tol=0.005).fit(data_train_normal)
W = ica.components_

# sort the ICs in importance order using L2 norm of each row
L2_norm = np.linalg.norm(W, 2, axis=1)
sort_order = np.flip(np.argsort(L2_norm)) # descending order
L2_norm_sorted_pct = 100*L2_norm[sort_order]/np.sum(L2_norm)

plt.figure()
plt.plot(L2_norm, 'b'), plt.xlabel('IC number (unsorted)'), plt.ylabel('L2 norm')
plt.figure()
plt.plot(L2_norm_sorted_pct, 'b+'), plt.xlabel('IC number (sorted)'), plt.ylabel('% L2 norm')

# re-arrange rows of W matrix
W_sorted = W[sort_order,:] # row 1 now corresponds to the most important IC and so on
```

`

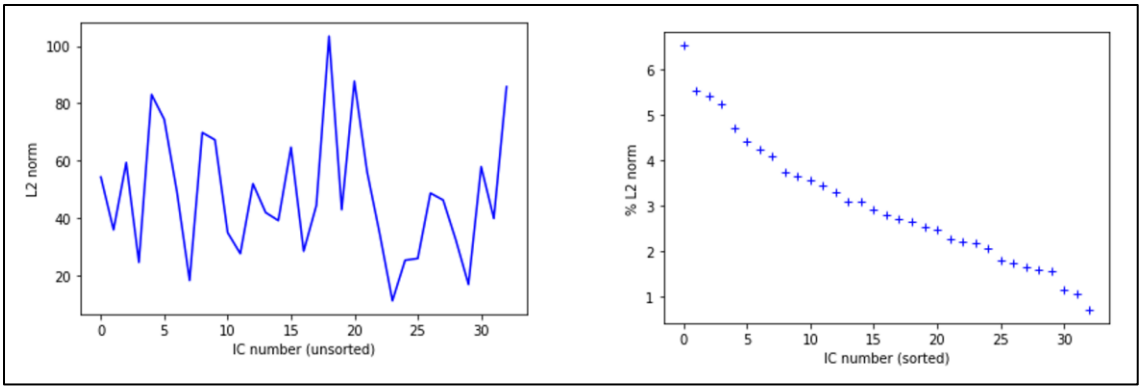

Figure 6.4: (Unsorted) L_2 norm of each row of $\boldsymbol{W}$ and (sorted) percentage of the L_2 norms

After the ordering of the ICs, 2 approaches could be utilized to determine the optimal number of ICs. Approach 1 uses the sorted L_2 norm plot to determine a cut-off IC number beyond which the norms are relatively insignificant. For our dataset, as seen in Figure 6.4, no clear cut-off number is apparent. Another approach entail choosing the number of ICs equal to the number of PCs. This approach also ensures fair comparison between ICA and PCA. We will use this 2nd approach in this chapter.

```
# decide # of ICs to retain via PCA variance method and compute ICs
from sklearn.decomposition import PCA
pca = PCA().fit(data_train_normal)

explained_variance = 100*pca.explained_variance_ratio_ # in percentage
cum_explained_variance = np.cumsum(explained_variance) # cumulative % variance explained

n_comp = np.argmax(cum_explained_variance >= 90) + 1
print('Number of PCs cumulatively explaining atleast 90% variance: ', n_comp)

# compute ICs with reduced dimension
Wd = W_sorted[0:n_comp,:]
Sd = np.dot(Wd, data_train_normal.T) # row 1 contains scores of the most important IC

>>> Number of PCs cumulatively explaining atleast 90% variance:  17
```

Note that the FastICA.fit method expects each row of data matrix ($\boldsymbol{X}$) to represent a sample while we used a transposed form in our mathematical descriptions. This may cause confusion at times. Nonetheless, the shapes of the extracted mixing, demixing, and whitening matrices are same in both the places.

`

6.2 Process Monitoring via ICA for Tennessee Eastman Process

The illustration example in Figure 6.1 showed that if the latent variables are non-Gaussian distributed then ICA can extract the latent signals better than PCA. For such systems, it can be expected that ICA-based monitoring will give better performance than PCA-based monitoring. Like PCA/PLS-based monitoring mechanism, monitoring metrics and their corresponding thresholds are computed using normal/fault-free operation data. The metric values for test data are compared against the thresholds to check for presence of faults.

Fault detection indices

Three monitoring metrics ($I^2,\ SPE,\ I_e^2$) are computed using training data. Note that because the ICs are independent, it is technically a reasonable approach to directly monitor the ICs separately using univariate statistical process control (SPC) techniques. However, monitoring multiple ICs simultaneously can be inconvenient. The first statistic, I^2, is defined as the sum of the squared independent scores (in reduced dimension space) and is a measure of systematic process variations (like PCA T^2 statistic). Let $\boldsymbol{s}_i$ denote the i^{th} column of matrix $\boldsymbol{S}_d$ which represents the i^{th} sample/data-point in the reduced IC space. The I^2 index for this sample is calculated by

$$I^2 = \boldsymbol{s}_i^T \boldsymbol{s}_i \qquad \text{eq. 6}$$

You may wonder why we have not included the covariance matrix as we did for T^2 metric computation. This is because the variance of each IC score is same (remember, the L_2 norm is same) and thus inclusion of covariance matrix is redundant.

The second index, SPE, represents the distance between the measured and reconstructed data-point in the measurement space. For its computation, let us construct a matrix $\boldsymbol{B}_d$ by selecting the columns from matrix $\boldsymbol{B}$ whose indices correspond to the indices of the rows selected from $\boldsymbol{W}$ when we generated matrix $\boldsymbol{W}_d$. Let $\boldsymbol{x}_i$ and $\hat{\boldsymbol{x}}_i$ denote the i^{th} measured and re-constructed sample. Then,

$$\begin{aligned} \hat{\boldsymbol{x}}_i &= \boldsymbol{Q}^{-1}\boldsymbol{B}_d\boldsymbol{s}_i = \boldsymbol{Q}^{-1}\boldsymbol{B}_d\boldsymbol{W}_d\boldsymbol{x}_i \\ \boldsymbol{e}_i &= \boldsymbol{x}_i - \hat{\boldsymbol{x}}_i \\ SPE &= \boldsymbol{e}_i^T\boldsymbol{e}_i \end{aligned} \qquad \text{eq. 7}$$

The third metric, I_e^2, is based on the excluded ICs. Let $\boldsymbol{W}_e$ contain the excluded rows of matrix $\boldsymbol{W}$. Then

$$\boldsymbol{s}_i^e = \boldsymbol{W}_e\boldsymbol{x}_i$$

`

$$I_e^2 = (\boldsymbol{s}_i^e)^T \boldsymbol{s}_i^e \qquad \text{eq. 8}$$

Due to the non-Gaussian nature of the ICs, we do not have convenient closed-form equations for computing the thresholds/control limits of the above computed indices. A standard practice is to use Kernel density Estimation (KDE) for threshold determination of ICA metrics. Since we will learn KDE in a later chapter, we will employ the percentile method here.

Below we define a function that takes an ICA model and data matrix as inputs and returns the monitoring metrics. Figure 6.5 shows the monitoring charts along with 99% control limits.

```
# Define function to compute ICA monitoring metrics for training or test samples
def compute_ICA_monitoring_metrics(ica_model, number_comp, data):
    """
    data: numpy array of shape = [n_samples, n_features]

    Returns
    ----------
    monitoring_stats: numpy array of shape = [n_samples, 3]
    """
    n = data.shape[0]

    # model parameters
    W = ica.components_
    L2_norm = np.linalg.norm(W, 2, axis=1)
    sort_order = np.flip(np.argsort(L2_norm))
    W_sorted = W[sort_order,:]

    # compute I2
    Wd = W_sorted[0:number_comp,:]
    Sd = np.dot(Wd, data.T)
    I2 = np.array([np.dot(Sd[:,i], Sd[:,i]) for i in range(n)])

    # compute Ie2
    We = W_sorted[n_comp:,:]
    Se = np.dot(We, data.T)
    Ie2 = np.array([np.dot(Se[:,i], Se[:,i]) for i in range(n)])

    # compute SPE
    Q = ica.whitening_
    Q_inv = np.linalg.inv(Q)
```

`

```
    A = ica.mixing_
    B = np.dot(Q, A)
    B_sorted = B[:,sort_order]
    Bd = B_sorted[:,0:n_comp]

    data_reconstruct = np.dot(np.dot(np.dot(Q_inv, Bd), Wd), data.T)
    e = data.T - data_reconstruct
    SPE = np.array([np.dot(e[:,i], e[:,i]) for i in range(n)])

    monitoring_stats = np.column_stack((I2, Ie2, SPE))
    return monitoring_stats

# Define a couple of helper functions
def draw_monitoring_chart(values, CL, yLabel):
    plt.figure()
    plt.plot(values)
    plt.axhline(CL, color = "red", linestyle = "--")
    plt.xlabel('Sample #'), plt.ylabel(yLabel)

def draw_ICA_monitoring_charts(ICA_statistics, CLs, trainORtest):
    """ draw monitoring charts for given data

    parameters
    ----------
    ICA_statistics: numpy array of shape = [n_samples, 3]
    CLs: List of control limits
    trainORtest: 'training' or 'test'
    """

    # I2 chart, Ie2 chart, SPE chart
    draw_monitoring_chart(ICA_statistics[:,0], CLs[0], 'I2 for ' + trainORtest + ' data')
    draw_monitoring_chart(ICA_statistics[:,1], CLs[1], 'Ie2 for ' + trainORtest + ' data')
    draw_monitoring_chart(ICA_statistics[:,2], CLs[2], 'SPE for ' + trainORtest + ' data')

# Draw monitoring charts for training data
ICA_statistics_train = compute_ICA_monitoring_metrics(ica, n_comp, data_train_normal)
I2_CL = np.percentile(ICA_statistics_train[:,0], 99)
Ie2_CL = np.percentile(ICA_statistics_train[:,1], 99)
SPE_CL = np.percentile(ICA_statistics_train[:,2], 99)

draw_ICA_monitoring_charts(ICA_statistics_train, [I2_CL, Ie2_CL, SPE_CL], 'training')
```

`

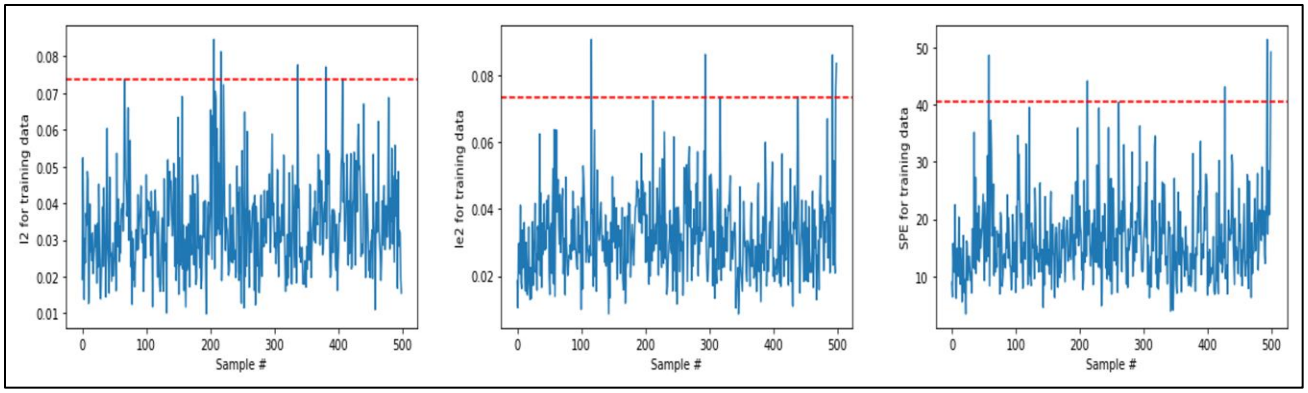

Figure 6.5: ICA monitoring charts for TEP training data

Fault detection for test data

Lee et al.[29] and Yin et al.[30] have published detailed accounts of process monitoring performances of several ML techniques for TEP and have shown superior fault detection capability of ICA. We will try to corroborate their findings for some of the faults. But let us first learn a few terms that are commonly used to evaluate performance of any monitoring tool. The terms are fault detection rate (FDR) and false alarm rate (FAR).

$$\text{FDR} = \frac{\text{Number of samples correctly flagged as faulty}}{\text{Total number of faulty samples}} \times 100$$

$$\text{FAR} = \frac{\text{Number of samples incorrectly flagged as faulty}}{\text{Total number of non} - \text{faulty samples}} \times 100$$

A well-designed monitoring tool has high FDR and low FAR values. Low FDR and high FAR values are both undesirable because while low FDR value leads to safety issues and economic losses due to delayed/no abnormality detection, high FAR value leads to loss of user's confidence in the tool and eventually tool's demise.

Let us now use these terms to compare the performances of PCA and ICA for detecting Faults 10 and 5. Figures 6.6 and 6.7 show the ICA/PCA monitoring charts for these faults. For Fault 10, ICA gives significantly higher FDR. There are many samples between 400 to 600 where PCA metrics are below their control limits despite the presence of process abnormalities. The performance difference is even more prominent for Fault 5 test data. Sample 400 onwards PCA charts incorrectly indicate a normal operation although it is known that the faulty condition remains till the end of the sampling period. Another point to note is that both PCA

[29] Lee et al., Statistical monitoring of dynamic processes based on dynamic independent component analysis, Chemical Engineering Science, 2004

[30] Yin et al., A comparison study of basic data-driven fault diagnosis and process monitoring methods on the benchmark Tennessee Eastman process, Journal of Process Control, 2012

`

and ICA have low FAR values as not many samples before sample 160 violate the control limits. Here again ICA has lower/better FAR values. Similar FAR behavior is observed for the non-faulty test dataset. We have not shown FAR values, but you should compute them and confirm that ICA gives lower FAR values.

```
# define function to compute FDR or FAR
def compute_alarmRate(monitoring_stats, CLs):
    """ calculate alarm rate

    parameters
    ----------
    monitoring_stats: numpy array of shape = [n_samples, 3]
    CLs: List of control limits

    Returns
    ----------
    alarmRate: float
    """
    violationFlag = monitoring_stats > CLs
    alarm_overall = np.any(violationFlag, axis=1) # violation of any metric => alarm
    alarmRate = 100*np.sum(alarm_overall)/monitoring_stats.shape[0]

    return alarmRate

# fetch test data and select data as done during training
TEdata_Fault_test = np.loadtxt('d10_te.dat')

xmeas = TEdata_Fault_test[:,0:22]
xmv = TEdata_Fault_test[:,41:52]
data_Fault_test = np.hstack((xmeas, xmv))

# scale data
data_test_scaled = scaler.transform(data_Fault_test)

# compute statistics and draw charts
ICA_statistics_test = compute_ICA_monitoring_metrics(ica, n_comp, data_test_scaled)
draw_ICA_monitoring_charts(ICA_statistics_test, [I2_CL, Ie2_CL, SPE_CL], 'test')

# compute FAR or FDR
alarmRate = compute_alarmRate(ICA_statistics_test[160:,:], [I2_CL, Ie2_CL, SPE_CL])
                                                    # fault starts sample 160 onwards
```

`

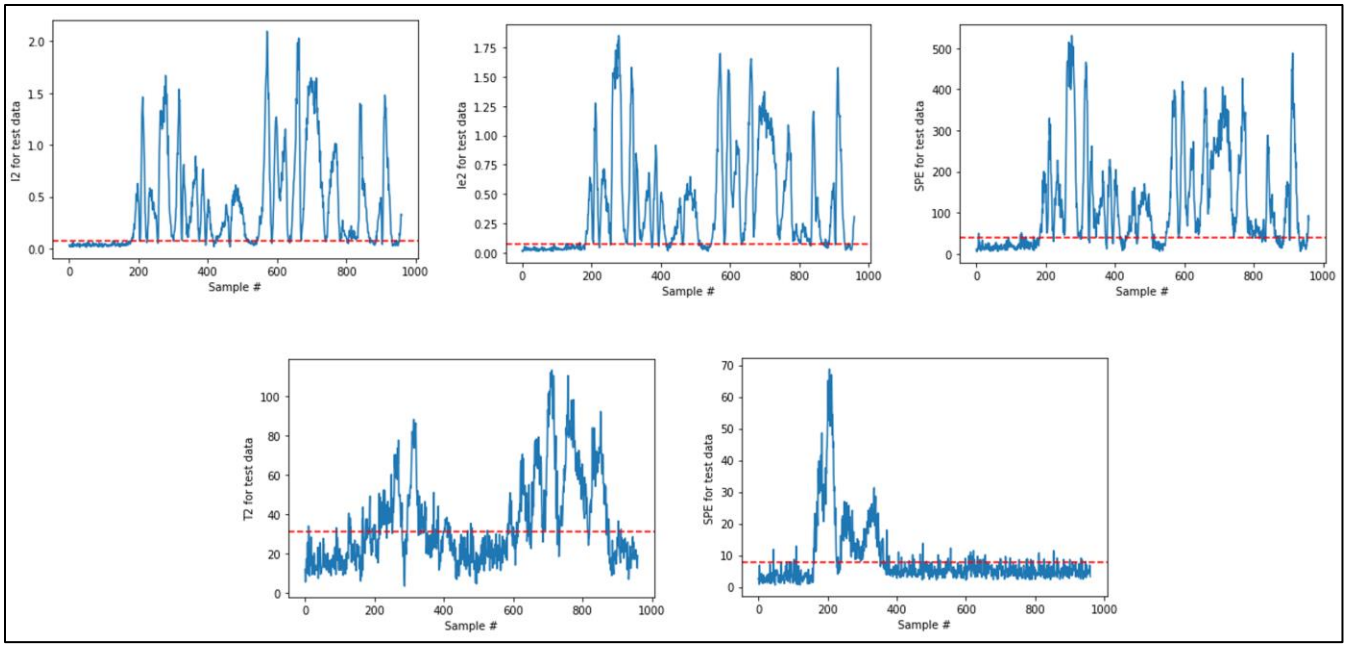

Figure 6.6: Monitoring charts for Fault 10 data with ICA (top, FDR = 90.8%) and PCA (bottom, FDR = 75.6%) metrics

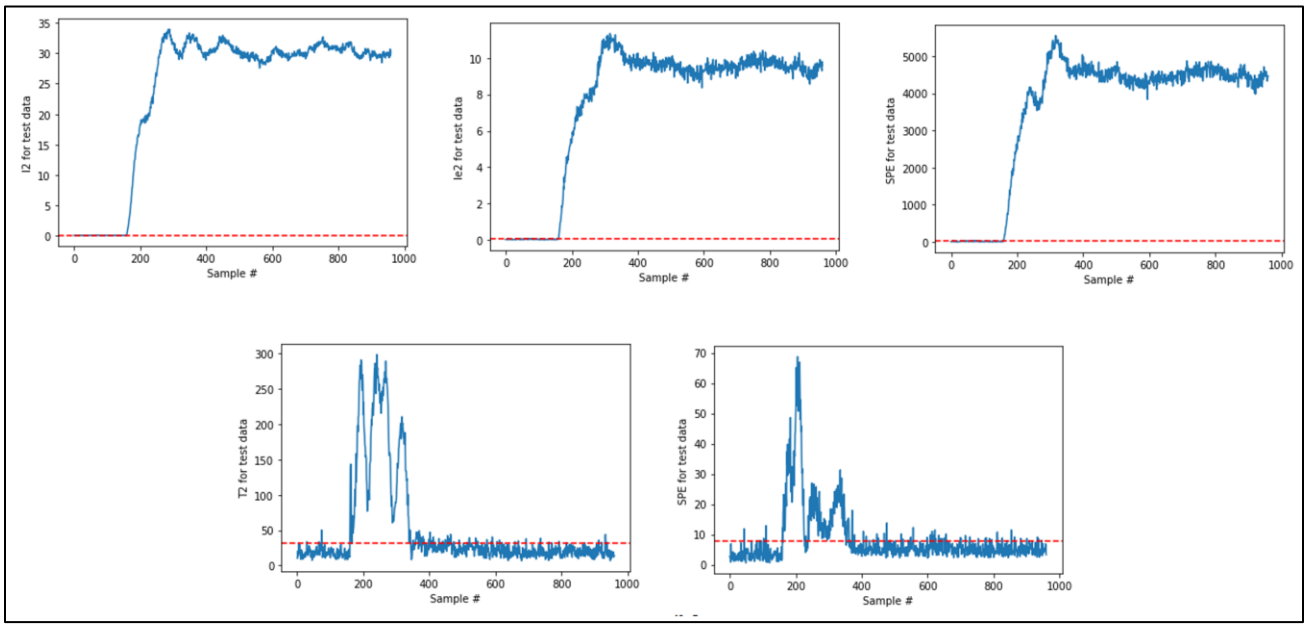

Figure 6.7: Monitoring charts for Fault 5 data with ICA (top, FDR = 100%) and PCA (bottom, FDR = 41.5%) metrics

Although PCA gave poor performance compared to ICA for the 2 test datasets, there is no general criteria that we could have used to predict this. The monitoring performance depends on the specific system under study and trial-and-error method is frequently employed for selection of a monitoring algorithm.

`

Like PCA and PLS, nonlinear and dynamic variants of ICA have also been developed. The reader is encouraged to see the works of Lee et al.[28] for dynamic ICA and Lee et al.[31] for nonlinear ICA.

6.3 FDA: An Introduction

Fisher Discriminant Analysis (FDA), also called linear discriminant analysis (LDA), is a multivariate dimensionality reduction technique which maximizes the 'separation' in the lower dimensional space between data belonging to different classes. For conceptual understanding, consider the simple illustration in Figure 6.8 where a 2D dataset (with 2 classes of data) has been projected onto 1D spaces by FDA and PCA. The respective 1-D scores show that while the two classes of data are well segregated in LD space, the segregation is very poor in PC space. This observation was expected because PCA, while determining the projection directions, does not consider information about different data classes. Therefore, if your intention is to reduce dimensionality for subsequent data classification and training data is labeled into different classes, then FDA is more suitable.

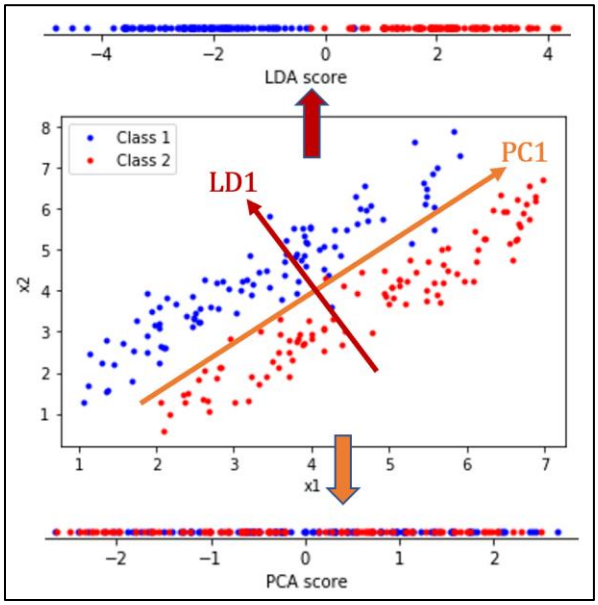

Figure 6.8: Simple illustration of FDA vs PCA. The arrows in the x1 vs x2 plot show the direction vectors of 1st components of the corresponding methods

Due to the powerful data discrimination ability of FDA, it is widely used in process industry for operating mode classification and fault diagnosis. Large-scale industrial processes often

[31] Lee et al., Fault detection of non-linear processes using kernel independent component analysis, The Canadian Journal of Chemical Engineering, 2007

`

experience different kinds of process issues/faults and it is imperative to accurately identify the specific issue quickly during online operations to minimize economic losses. During model training, FDA learns from data collected from historical process failures (data from a specific process fault form a class) to find the optimal projection directions and classify abnormal process data into specific faults in real-time. We will study one such application in this chapter.

Mathematical background

To facilitate data classification, FDA not only maximizes the separation between the classes but also minimizes the variation/scatter within each class. To see how this is achieved, let us first consider a dataset matrix $\boldsymbol{X} \in \mathbb{R}^{N \times \mathrm{m}}$ consisting of *N* samples, N_1 of which belong to class 1 (ω_1) and N_2 belong to class 2 (ω_2). FDA seeks to find a projection vector $\boldsymbol{w} \in \mathbb{R}^{\mathrm{m}}$ such that the projected scalars/samples ($z = \boldsymbol{w}^T\mathbf{x}$) are maximally separated. Let $\tilde{\mu}_1$and $\tilde{\mu}_2$ denote the means of projected values of classes 1 and 2, respectively

$$\tilde{\mu}_i = \frac{1}{N_i} \sum_{z \in \omega_1} z$$

Class separation could be quantified as distance between the projected means $|\tilde{\mu}_1 - \tilde{\mu}_2|$; this, however, is not a robust measure as shown by the illustration below.

(a) Means farther but poor class separability due to large variance

(b) Means closer but better class separability due to small variance

To quantify the variability within class ω_i we define a term called scatter, $\tilde{s}_i^2$

$$\tilde{s}_i^2 = \sum_{z \in \omega_1} (z - \tilde{\mu}_i)^2$$

The separation criterion is now defined as the normalized distance between the projected means. This formulation seeks to find a projection where the class means are far apart and samples from the same class are close to each other.

$$J(\boldsymbol{w}) = \frac{|\tilde{\mu}_1 - \tilde{\mu}_2|^2}{\tilde{s}_1^2 + \tilde{s}_2^2}$$

`

Using the base relation $y = \boldsymbol{w}^T\mathbf{x}$ and straightforward algebraic manipulations[32], one can equivalently represent the objective $J(\boldsymbol{w})$ as follows which also holds for any (*p*) number of data classes

$$J(\boldsymbol{w}) = \frac{\boldsymbol{w}^T\boldsymbol{S}_b\boldsymbol{w}}{\boldsymbol{w}^T\boldsymbol{S}_w\boldsymbol{w}} \qquad \text{eq. 9}$$

$\boldsymbol{S}_b$ (between-class-scatter matrix) and $\boldsymbol{S}_w$ (within-class-scatter matrix) are defined as

$$\boldsymbol{S}_b = \sum_{j=1}^{p} N_j\,(\boldsymbol{\mu}_j - \boldsymbol{\mu})\,(\boldsymbol{\mu}_j - \boldsymbol{\mu})^T$$

$$\boldsymbol{S}_w = \sum_{j=1}^{p} \boldsymbol{S}_j$$

$$\boldsymbol{S}_j = \sum_{\boldsymbol{x}_i \in \omega_1} (\boldsymbol{x}_i - \boldsymbol{\mu}_j)(\boldsymbol{x}_i - \boldsymbol{\mu}_j)^T$$

where, $\boldsymbol{\mu} \in \mathbb{R}^{\mathrm{m}}$ and $\boldsymbol{\mu}_j \in \mathbb{R}^{\mathrm{m}}$ denote the mean vectors of all the N samples and N_j samples from j^{th} class, respectively, in the measurement space. The first FDA vector, $\boldsymbol{w_1}$, is found by maximizing *J*(**w**) and the subsequent vectors are found by solving the same problem with the added constraints of orthogonality to previously computed vectors. Note that there can be at most *p*-1 FDA vectors. Alternatively, like PCA, the vectors can also be computed as solutions of a generalized eigenvalue problem

$$\boldsymbol{S}_w^{-1}\boldsymbol{S}_b\boldsymbol{w} = \lambda\boldsymbol{w} \qquad \textit{eq. 10}$$

where λ = *J*(**w**). Therefore, the eigenvalues (λ_s) indicate the degree of separability among the data classes when projected onto the corresponding eigenvectors. The first discriminant/FDA vector/eigenvector corresponds to the largest eigenvalue, the 2nd FDA vector is associated with the 2nd largest eigenvalue, and so on. Once the FDA vectors are determined, data-points can be projected, and classification models can be built in the reduced FDA space. Overall, FDA transformation from m dimensional space to *p*-1 dimensional FDA space can be represented as

$$\boldsymbol{Z} = \boldsymbol{X}\boldsymbol{W}_p$$

where $\boldsymbol{W}_p \in \mathbb{R}^{m\times(\mathrm{p}-1)}$ contains the *p*-1 FDA vectors as columns and $\boldsymbol{Z} \in \mathbb{R}^{N\times(\mathrm{p}-1)}$ is the data-matrix in the transformed space where each row is the transformed sample. The transformed samples are optimally separated in the FDA space.

[32] Elhabian & Farag, A tutorial on data reduction Linear Discriminant Analysis, September 2009

`

Dimensionality reduction for Tennessee Eastman Process

To demonstrate the dimension reduction and class separability capabilities of FDA, let's use data from 3 fault classes (faults 5, 10, and 19) in the TEP dataset. With 3 fault classes, FDA will result in maximum 2 transformed dimensions. We will also perform PCA for comparison.

```
# fetch TEP data for faults 5,10,19
TEdata_Fault5_train = np.loadtxt('d05.dat')
TEdata_Fault10_train = np.loadtxt('d10.dat')
TEdata_Fault19_train = np.loadtxt('d19.dat')
TEdata_Faulty_train = np.vstack((TEdata_Fault5_train, TEdata_Fault10_train,
TEdata_Fault19_train))

# select variables (discarding composition measurements)
xmeas = TEdata_Faulty_train[:,0:22] # 22 continuous process variables
xmv = TEdata_Faulty_train[:,41:52] # 11 manipulated variables
data_Faulty_train = np.hstack((xmeas, xmv))

# generate sample labels
n_rows_train = TEdata_Fault5_train.shape[0]
y_train = np.concatenate((5*np.ones(n_rows_train,), 10*np.ones(n_rows_train,),
19*np.ones(n_rows_train,)))

# scale data
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
Faultydata_train_scaled = scaler.fit_transform(data_Faulty_train)

# fit LDA model
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
lda = LinearDiscriminantAnalysis()
scores_train_lda = lda.fit_transform(Faultydata_train_scaled, y_train)

# visualize LDA scores
plt.figure()
plt.plot(scores_train_lda[0:n_rows_train,0], scores_train_lda[0:n_rows_train,1], 'b.', label='Fault 5')
plt.plot(scores_train_lda[n_rows_train:2*n_rows_train,0],
        scores_train_lda[n_rows_train:2*n_rows_train,1], 'r.', label='Fault 10')
plt.plot(scores_train_lda[2*n_rows_train:3*n_rows_train,0],
        scores_train_lda[2*n_rows_train:3*n_rows_train,1], 'm.', label='Fault 19')
plt.legend()
```

`

Figure 6.9 shows the transformed samples in 2 dimensions after FDA and PCA. FDA is able to provide a clear separation of Fault 5 samples; however, it could not separate Faults 10 and 19 data. Infact, the 2nd discriminant (FD2) contributes little to the discrimination and therefore, only FD1 is needed to separate samples from Fault 5. To segregate Faults 10 and 19, kernel FDA can be explored[33]. Linear PCA, on the other hand, fails to separate any of the classes.

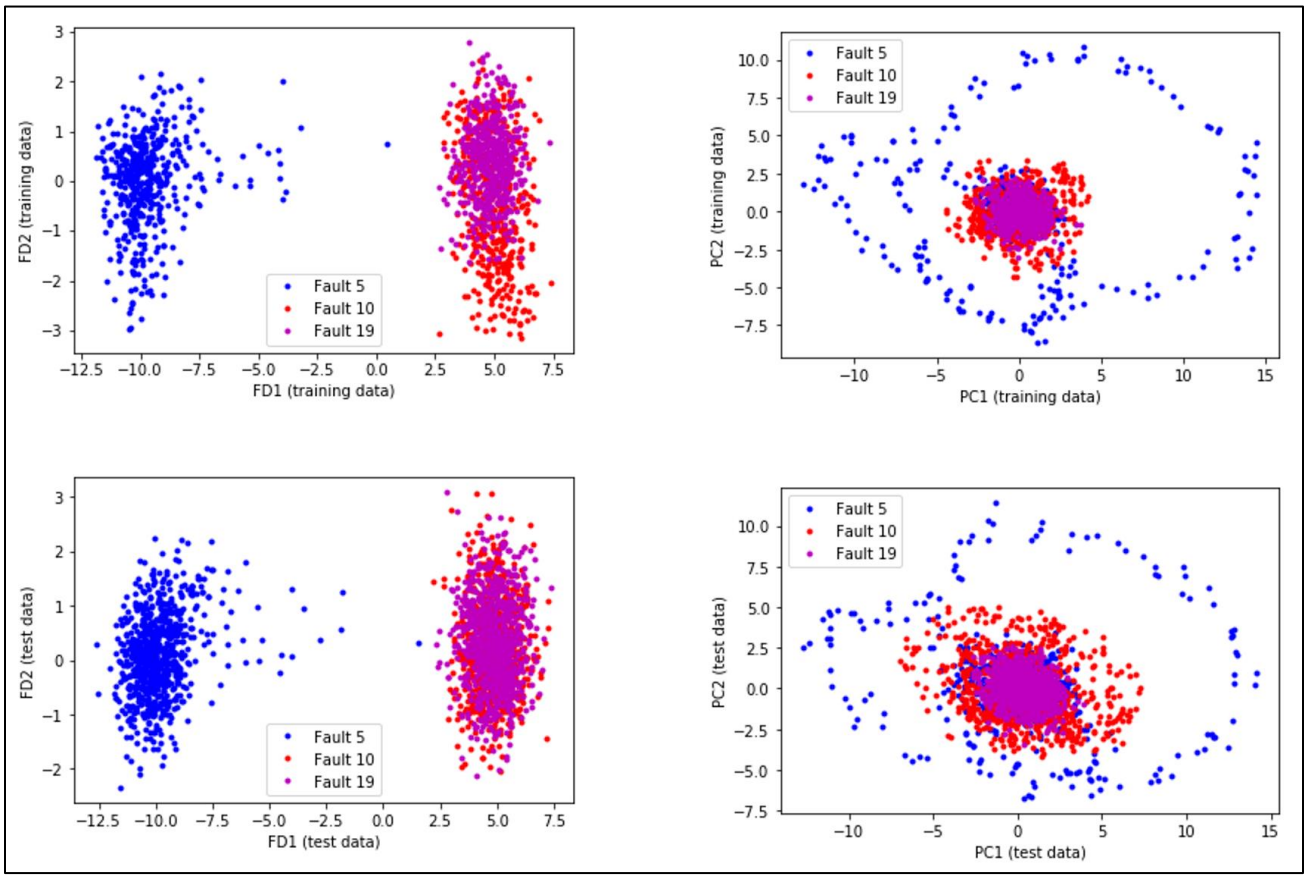

Figure 6.9: FDA and PCA scores in 2 dimensions with 3 fault classes from TEP training (top) and test (bottom) dataset

While FDA is a powerful tool for fault diagnosis, it can also be used for fault detection by including data from 'no-fault' or normal plant operation as a separate class.

[33] Hyun-Woo Cho, Nonlinear feature extraction and classification of multivariate process data in kernel feature space, Expert Systems with Applications, 2007

`

6.4 Fault Classification via FDA for Tennessee Eastman Process

After projecting the samples onto the FDA space, any classification technique can be chosen to classify or diagnose the specific fault. A popular T^2 statistic-based approach entails computing a T^2 control limit for each fault class using the training data. The T^2 control limit ($\boldsymbol{T^2_{CL,j}}$) for the j^{th} fault class represents a boundary around the projected samples from the j^{th} fault in the lower-dimensional space – any given sample lying inside the boundary belongs to the j^{th} fault class. Mathematically, this can be specified as

$$\boldsymbol{T^2_{sample,j} < T^2_{CL,j} \Rightarrow sample\ belongs\ to\ the\ j^{th}\ fault\ class}$$

$\boldsymbol{T^2_{sample,j}}$ for a sample for the j^{th} class is given by

$$\boldsymbol{T^2_{sample,j} = (z_{sample} - \tilde{\mu}_j)^T \tilde{S}_j (z_{sample} - \tilde{\mu}_j)}$$

where, $\boldsymbol{\tilde{\mu}_j}$ and $\boldsymbol{\tilde{S}_j}$ denote the mean and covariance matrix of the projected samples belonging to the j^{th} fault class in training dataset, respectively, and $\boldsymbol{z_{sample}}$ denotes the projected sample in the FDA space. $T^2_{CL,j}$ can be obtained using the same expression we used in PCA, $\frac{k(N_j^2-1)}{N_j(N_j-k)} F_{k,N-k}(\alpha)$, where k denotes the number of dimensions retained in the FDA space. For illustration, let us see how many samples from the Fault 5 test data get correctly identified.

```
# compute control limit
import scipy.stats
Nj = n_rows_train
k = 2

alpha = 0.01 # 99% control limit
T2_CL = k*(Nj**2-1)*scipy.stats.f.ppf(1-alpha,k,Nj-k)/(Nj*(Nj-k))

# mean and covariance for Fault 5 class
scores_train_lda_Fault5 = scores_train_lda[0:n_rows_train,:]
cov_scores_train_Fault5 = np.cov(scores_train_lda_Fault5.T)
mean_scores_train_Fault5 = np.mean(scores_train_lda_Fault5, axis = 0)

# fetch TE test data for fault 5
TEdata_Fault5_test = np.loadtxt('d05_te.dat')
```

`

```
TEdata_Fault5_test = TEdata_Fault5_test[160:,:] # faulty operation start sample 160 onwards
n_rows_test = TEdata_Fault5_test.shape[0]
xmeas = TEdata_Fault5_test[:,0:22]
xmv = TEdata_Fault5_test[:,41:52]
data_Faulty_test = np.hstack((xmeas, xmv))

# scale test data and transform
Faultydata_test_scaled = scaler.transform(data_Faulty_test)
scores_test_lda = lda.transform(Faultydata_test_scaled)

# compute T2 statistic for test data for Fault 5 class
T2_test = np.zeros((n_rows_test,))
for sample in range(n_rows_test):
    score_sample = scores_test_lda[sample,:]
    score_sample_centered = score_sample - mean_scores_train_Fault5
    T2_test[sample] = np.dot(np.dot(score_sample_centered[np.newaxis,:],
                      np.linalg.inv(cov_scores_train_Fault5)), score_sample_centered[np.newaxis,:].T)

# plot test prediction
outsideCL_flag = T2_test > T2_CL
insideCL_flag = T2_test <= T2_CL
plt.figure()
plt.plot(scores_test_lda[outsideCL_flag,0], scores_test_lda[outsideCL_flag,1], 'k.', label='within
        Fault 5 boundary')
plt.plot(scores_test_lda[insideCL_flag,0], scores_test_lda[insideCL_flag,1], 'b.', label='outside
        Fault 5 boundary')
plt.xlabel('FD1 (test data)')
plt.ylabel('FD2 (test data)')
```

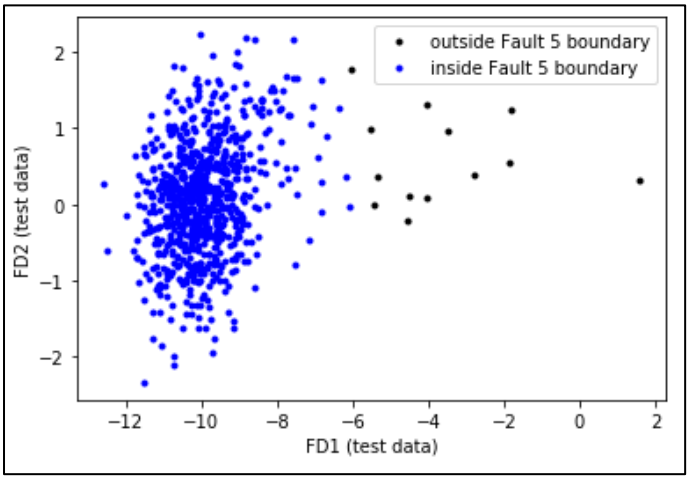

Figure 6.10: Classification result for TEP Fault 5 test samples

`

About 98% of the samples have been correctly identified as belonging to Fault 5. As shown in Figure 6.10, some of the samples which fall far away from the mean violate the $T^2_{CL,5}$ and therefore are not classified as Fault 5.

Process Monitoring Terminology

In our process monitoring applications so far, we have used terms like fault detection, fault diagnosis, and fault classification. You may, however, also encounter other terms like fault identification, fault isolation. While these terms may get used interchangeably, there exist nuanced differences. Fault detection refers to the task of determining whether abnormal process conditions have occurred. Fault identification and fault isolation, both refer to the task of identifying the process variables that exhibit abnormal behavior. Identifying these variables can help in determining the root-cause of the fault. The step of 'fault diagnosis' in the PCA chapter should have been called 'fault identification'!

Fault diagnosis refers to the task of finding which specific fault has occurred or determining the root-cause of the fault. Fault classification falls in this category. While fault isolation is non-supervised, fault diagnosis is supervised.

Summary

With this chapter, we have now covered all the popular classical dimensionality reduction techniques that are frequently utilized for analyzing process data. This chapter has also provided an important message: blind application of a single technique all the time may not yield best results. The techniques should be chosen according to the process system (Gaussian vs non-Gaussian) and objective (fault detection vs fault classification) at hand. ICA and FDA are powerful techniques and there lies much more to them that what we have touched in this chapter. While you are encouraged to explore more about these methods (now that you have conceptual understandings), we will move to study another powerful ML technique called Support Vector Machines in the next chapter.